

**SCHEDULING OF SETS OF TASKS ON
SEQUENT SYMMETRY S/81:
AN EMPIRICAL STUDY**

By

AMIR ALI THOBANI

Bachelor of Engineering

N.E.D. University of Engineering and Technology

Karachi, Pakistan

1990

**Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1993**

SCHEDULING OF SETS OF TASKS ON
SEQUENT SYMMETRY S/81:
AN EMPIRICAL STUDY

Thesis Approved:

M. Samadzadeh-H.

Thesis Adviser

Blayne E. Mayfield

Huizhu Lu

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

First of all, I would like to thank my thesis advisor Dr. Mansur H. Samadzadeh for his continuous guidance, dedication, and valuable instruction throughout this research work. Without his encouragement and motivation, completion of this thesis would not have been possible.

Special thanks are due to Dr. Farideh A. Samadzadeh for her advice, guidance, and encouragement during the experimental part of my thesis. Her observations and findings from the experiments' data helped me a lot.

I would also like to thank Drs. Blayne Mayfield and Huizhu Lu for their suggestions and advice while serving on my thesis committee. In addition, I would like to thank my supervisor at the University Computer Center, Mr. Larry P. Watkins, for allowing me to follow a flexible work schedule during my thesis research.

Finally, I wish to thank my family, especially my mother Sultana and my younger brother Ashiq Ali. It was their continuous support that gave me the motivation and inspiration to complete my graduate studies.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. MULTIPROCESSING	4
2.1 Shared-Memory Multiprocessors	5
2.2 Distributed-Memory Multiprocessors	5
2.3 Heterogeneous Multiprocessing	6
2.4 Homogeneous Multiprocessing	7
III. SCHEDULING	8
3.1 System-Level Scheduling	9
3.2 User-Level Scheduling	9
3.3 Static Scheduling	10
3.4 Dynamic Scheduling	11
3.5 Task System	12
IV. STATIC SCHEDULING OF INDEPENDENT TASKS	13
4.1 Experimentation Platform	13
4.2 Objective of the Experiment	14
4.3 Detail of the Experiment	14
4.3.1 Description of the Tasks	14
4.3.2 Task Category 1 (User Program)	15
4.3.3 Task Category 2 (System Command)	16
4.3.4 Description of the Eight Sets of Tasks	16
4.3.5 Static Scheduling of Tasks	20
4.4 Analysis of the Results	20
4.4.1 Task Set One (19, 100, WHILE, INCR)	21
4.4.2 Task Set Two (19, 100, WHILE, DECR)	22
4.4.3 Task Set Three (19, 100, WHILE, EQUAL)	23
4.4.4 Task Set Four (19, 100, WHILE, RAND)	23
4.4.5 Task Set Five (19, 100, SLEEP, INCR)	23
4.4.6 Task Set Six (19, 100, SLEEP, DECR)	24
4.4.7 Task Set Seven (19, 100, SLEEP, EQUAL)	24
4.4.8 Task Set Eight (19, 100, SLEEP, RAND)	25
4.5 Summary of Preliminary Result	25
V. HEURISTIC SCHEDULING OF TASKS	27
5.1 Notation and Definitions	27
5.2 Scheduling of Independent Task Systems	29
5.2.1 Variant-Load Algorithm	30

Chapter	Page
5.2.2 Divide and Fold Algorithm	31
5.3 Scheduling of Dependent Task Systems	31
5.3.1 Ranked Weight Algorithm	32
5.3.2 ESP/VL Algorithm.	33
5.4 Task System Generator	34
5.5 Scheduling Methodology	34
5.5.1 Self-Timed Scheduling	35
5.5.2 Instrumentation	36
5.6 Task System Characteristics Used in the Simulation	37
5.6.1 Independent Task Systems	38
5.6.2 Dependent Task Systems	38
5.7 Results of the Experiment	39
5.7.1 Execution of Independent Task Systems	40
5.7.2 Execution of Dependent Task Systems	43
5.8 Summary and Conclusions	48
VI. SUMMARY AND FUTURE WORK	50
6.1 Introduction	50
6.2 Summary.	50
6.3 Future Work	52
REFERENCES	54
APPENDICES	57
APPENDIX A - GLOSSARY AND TRADEMARK INFORMATION	57
APPENDIX B - GRAPHS	61
APPENDIX C - PROGRAM LISTING	70

LIST OF TABLES

Table	Page
I. STATIC SCHEDULING OF 100 TASKS ON 19 PROCESSORS	21
II. RESULT OF TEST 1 FOR SCHEDULING OF INDEPENDENT TASK SYSTEM USING VARIANT-LOAD ALGORITHM	41
III. RESULT OF TEST 2 FOR SCHEDULING OF INDEPENDENT TASK SYSTEM USING VARIANT-LOAD ALGORITHM	42
IV. RESULT OF TEST 1 FOR SCHEDULING OF INDEPENDENT TASK SYSTEM USING DIVIDE AND FOLD ALGORITHM	43
V. RESULT OF TEST 2 FOR SCHEDULING OF INDEPENDENT TASK SYSTEM USING DIVIDE AND FOLD ALGORITHM	43
VI. RESULT OF TEST FOR SCHEDULING OF DEPENDENT TASK SYSTEM USING RANKED WEIGHT ALGORITHM	46
VII. RESULT OF TEST FOR SCHEDULING OF DEPENDENT TASK SYSTEM USING ESP/VL ALGORITHM	47

LIST OF FIGURES

Figure	Page
1. Task Set One (19, 100, WHILE, INCR)	62
2. Task Set Two (19, 100, WHILE, DECR)	63
3. Task Set Three (19, 100, WHILE, EQUAL)	64
4. Task Set Four (19, 100, WHILE, RAND)	65
5. Task Set Five (19, 100, SLEEP, INCR)	66
6. Task Set Six (19, 100, SLEEP, DECR)	67
7. Task Set Seven (19, 100, SLEEP, EQUAL)	68
8. Task Set Eight (19, 100, SLEEP, RAND)	69

CHAPTER I

INTRODUCTION

As technology approaches physical limitations, parallel processor systems offer a promising and powerful alternative for high performance computing. With parallelism comes performance that could in principle be increased beyond any limit but, realistically speaking, parallelism has its limitations. These problems include building massively parallel machines with a large number of general-purpose processors and efficiently utilizing such massively parallel systems.

Performance of a parallel processor system, sometimes referred to as a multiprocessor system, does not depend only on the number of processors available in the system. Amdahl's law [Amdahl67] [Minsky70] presents arguments against increasing the number of processors beyond certain limits.

Once the number of processors in a system is fixed, the performance of the system depends generally on the detection of parallelism in the programs to be executed on the multiprocessor system. A program could be explicitly or implicitly divided into small programs or tasks, such that these tasks could be executed separately on different processors. The interdependencies among the tasks should indeed be considered before submitting them to processors. If parallelism is detected in a given program by dividing the program into small tasks, which could be executed on available processors, the

performance of the multiprocessor system could be increased by proper ordering (or scheduling) of these tasks to the available processors.

Task scheduling for multiprocessor systems could be done for both dependent and independent task systems. There are a number of algorithms developed in the literature for scheduling of independent and dependent tasks on multiprocessor system [Samadzadeh91, 92a, 92b] [Polychronopolous86].

Performance evaluation of scheduling algorithms on multiprocessor systems have generally been done in the literature using analytical models [Baer73]. Performance evaluation using analytical models is suitable for the comparative performance measurement of different algorithms, but it gives very little information about the actual improvement in performance of a particular multiprocessor system. On the other hand, if the performance evaluation is done using low-level simulation, where a set of tasks is executed on an actual system, then the result of the analysis gives information not only about the algorithms for the comparative analysis, but also about the behavior of the system and the system performance.

Scheduling could be done at the system level as well as at the user level. This thesis discusses the performance evaluation of the Sequent Symmetry S/81 system for the static scheduling at the user level for independent as well as dependent task systems using Variant-Load, Divide & Fold, Ranked Weight and ESP/VL algorithms [Samadzadeh91, 92a, 92b] [Polychronopolous86]. The key performance measure used for this study are the schedule length, performance ratio, efficiency, and speedup. Schedule length is also used in the performance evaluation for the analytical model [Samadzadeh91].

This study was done on a shared-memory multiprocessor machine, i.e., the Sequent Symmetry S/81 system running the DYNIX/ptx operating system. This multiprocessor system is available in the Computer Science Department at Oklahoma State University.

Chapter II of this thesis gives a brief description of different types of multiprocessors including shared-memory multiprocessors (used for this thesis). Chapter III presents a general discussion of scheduling. Different concepts in scheduling such as system-level and user-level scheduling are discussed. Chapter IV gives the description of an experiment performed to investigate the static scheduling of independent tasks on the Sequent Symmetry S/81 system. Performance evaluation and a brief discussion of different heuristic scheduling algorithms used for this thesis are presented in Chapter V. Finally, Chapter VI summarizes the main conclusions and elucidates some possible areas of future work.

CHAPTER II

MULTIPROCESSING

Multiprocessing in the sense of parallel processing can be defined as the simultaneous processing of two (or more) portions of the same program by two (or more) processing units [Baer73]. Multiprocessing is different from multiprogramming which is the time and resource sharing of two or more programs residing simultaneously in the primary memory. With multiprocessing, the throughput of the system is generally increased, the system becomes more reliable, and furthermore, the overhead of switching among jobs decreases as compared to a uniprocessing multiprogrammed environment.

Multiprocessors are typically general-purpose asynchronous parallel machines with multiple instruction-streams and multiple data-streams (MIMDs). They can be classified as being tightly-coupled or loosely-coupled [Sarkar89]. Processors in a tightly-coupled multiprocessor communicate through a shared memory and hence are called shared-memory multiprocessors. Processors in a loosely-coupled multiprocessor communicate by exchanging messages and generally have distributed memories, i.e., each of the processors has its own memory and hence such systems are called distributed-memory multiprocessors. Distributed-memory machines and sheared-memory machines could be further classified into homogeneous multiprocessors, heterogeneous multiprocessors, array processors, and vector processors [Baer73].

2.1 Shared-Memory Multiprocessors

The simplest form of a shared-memory machine is a shared-bus multiprocessor. A shared bus connects the processing elements to a global shared memory. Each processing element contains a processor and some local memory. Processors communicate via read and write accesses to the shared memory. Most designs use the local memory as a private cache, giving rise to the cache coherence problem [Archibald86]. A mechanism must exist to ensure that all private cache copies of a shared memory location are consistent. The cache coherence problem can be solved in software or hardware [Sarkar89]. The common solution (e.g., Xerox Dragon, DEC Firefly) is to use a hardware snooping cache controller for each cache, which monitors the bus transactions to maintain cache consistency. For efficiency, snooping cache controllers can also often satisfy a cache miss by accessing another cache instead of main memory. The Sequent Symmetry S/81 system, used for performance evaluation in this thesis, is a single bus shared-memory multiprocessor system.

2.2 Distributed-Memory Multiprocissors

Distributed-memory multiprocessors have no global shared memory. Instead, processors communicate by sending and receiving messages. The communication between processors is done over interconnection networks, and the performance of multiprocessor is primarily determined by the nature of the interconnection network. The simplest interconnection network for distributed-memory multiprocessors is a bus capable of handling interprocessor messages (i.e., a local area network). Some examples of interconnection networks includes star, tree, mesh, shuffle-exchange, and hypercube. The hypercube interconnection network has been used in at least

three multiprocessor designs - the Caltech Cosmic Cube, Intel iPSC, and NCUBE-10 [Sarkar89].

2.3 Heterogeneous Multiprocessing

In heterogeneous multiprocessing systems, the processing power of the processors in the system are not identical. Heterogeneous multiprocessing systems generally have distributed memories. One reason for heterogeneous multiprocessors having different processors is to employ processors of different processing power according to the processing needs. One example could be a large company with one head office, a couple of division offices, and a large number of branch offices. Since the branch offices would be responsible for data entry and data retrieval, a network of personal computers with one server would be enough. These branch offices would not require powerful multiprocessor machines. On the other hand, a division office is responsible for processing of data gathered from the branch offices and then sending the most important data to the main office. Hence, it can be argued that the division offices might need minicomputers of comparatively more processing power. On the other hand, the main office, where data from all the division offices is obtained and analyzed to obtain final reports for the top management of the company, will require powerful multiprocessors system. All these computers, i.e., personal computers at the branch offices, minicomputers at the division offices, and the mainframe in the main office communicate with each other with some kind of intercommunication network, and collectively they form a complete system. This system has processors of different powers and hence forms a heterogeneous multiprocessor system.

2.4 Homogeneous Multiprocessors

In homogeneous multiprocessor systems, the processors in the system are identical, all with the same processing power. Shared-memory multiprocessor systems generally have homogeneous multiprocessors. One example of a homogeneous multiprocessor system is the Sequent Symmetry S/81 system. Homogeneous multiprocessors systems are suitable for situations where all of the processing is done in the same place, i.e., the processing power is not geographically dispersed.

CHAPTER III

SCHEDULING

A Multiple Instruction Multiple Data (MIMD) parallel program is a collection of tasks that may run serially or in parallel. These tasks must be optimally placed on the processors of a particular machine if the shortest execution time is to be realized. This problem is known as the scheduling problem of parallel computing and has received considerable attention in recent years [Lewis92]. This problem is one of the challenging problems in parallel computing and is known to be NP-complete in its general form [Ullman75].

A set of resources and consumers provides a good example of a scheduling problem. Consumers need the service of one or more of the resources from the set of resources. The scheduling problem in this case is to find an efficient policy to manage the use of resources by the consumers in order to optimize some desired performance measure. Two examples of performance measures in the above case are the scheduling length and the mean time spent in the system by the consumers [Lewis92].

A schedule is the assignment of tasks to processors under time constraints [Polychronopolus88]. Schedule length is the time it takes to execute a given program under a specific scheduling algorithm. The execution time is calculated up to the moment the last processor working on that program finishes. For the same program, different scheduling algorithms in general yield different schedule lengths.

There has been a substantial amount of work done to decrease the schedule length of a set of tasks or jobs [Graham76] [Nutt92] [Sahni76] and [Samadzadeh91, 92a, 92b]. Given a number of tasks, the schedule length could be decreased by proper ordering or scheduling of the tasks. Of course, inter-dependencies among the tasks should be considered for correct sequencing. Depending on whether the user or the operating system is doing the scheduling of the tasks, scheduling can be classified as user level or system level scheduling.

3.1 System Level Scheduling

In scheduling at the system level, tasks are given to the operating system by a user and the operating system schedules the given tasks on the available processors [Thobani92]. Scheduling at the system level has the possible advantage that the operating system will presumably attempt to schedule the sets of tasks submitted by users in such a way that the overall schedule length of the tasks in the system is minimized. Since the users don't have any control over the scheduling of their tasks, it is not in general possible for a user to minimize the schedule length of a set of tasks by requesting that the operating system assign a higher priority to a particular set of tasks.

3.2 User Level Scheduling

Scheduling at the user level means that the users would schedule their tasks on the available processors. Thus, if there are n processors available in the system, a user should be able in principle to process n tasks simultaneously by scheduling one task on each processor (provided that the tasks are independent). Moreover, the user will also know which task is

running on which processor, which can be useful for reducing the schedule length. Once the set of tasks is obtained by the scheduler, these tasks could be scheduled on the available processors either statically or dynamically, giving rise to two further types of scheduling, namely, static scheduling and dynamic scheduling [Ha91] [Thobani92].

3.3 Static Scheduling

In static scheduling, the eligible tasks are assigned to processors at compile time. Then, at run-time, a run-time scheduler dispatches the tasks when the processor to which each task was assigned becomes available and all of the predecessor of that task have been executed [Ha91]. The algorithm for static scheduling determines which tasks a processor will perform, monitors the processing of the designated tasks by each processor, and finally enforces synchronization by ensuring that all processors finish their assigned tasks [SEQ92]. In another variation of static scheduling, the compiler determines the schedule for the tasks and assigns these tasks to the scheduled processors; then, at run-time, each processor waits for the predecessors of the next task in its queue to finish their execution and then executes that task. This type of scheduling is called *self-timed* scheduling [Ha91].

With static scheduling, information about a precedence-constrained task graph and the processing time of each task in the task graph must be known beforehand. This information provides the opportunity of applying different scheduling algorithms to decrease the schedule length. Hence, in static scheduling each task in a task graph has a static assignment to a

particular processor and, each time that task is submitted for execution, it is assigned to that processor [Lewis92].

The major advantage of static scheduling is that its run-time overhead is minimal [Polychronopolus88]. However, the disadvantage of static scheduling is its inadequacy in handling nondeterminacy in program execution [Lewis92].

3.4 Dynamic Scheduling

In dynamic scheduling, although the eligible tasks are scheduled by the scheduler at run-time, tasks are not divided amongst the processors in some predetermined way (as is the case with static scheduling). Instead, each processor checks for tasks at run-time by examining a task queue [SEQ92].

Dynamic scheduling is usually implemented as some kind of load balancing heuristic because the scheduler has only local information about a parallel program at any point in time. The disadvantage of dynamic scheduling is its inadequacy in finding global optimums, and the corresponding overhead that occurs because the schedule must be determined while the program is running [Lewis92].

The scheduling decision for dynamic scheduling is made at run-time, which tends to incur a penalty or overhead [Polychronopolus88]. Thus, the dynamic scheduling approach should be less sophisticated and rather simple. This overhead is the main disadvantage of dynamic scheduling. Excluding the overhead, it is not clear whether dynamic scheduling would be less effective than an optimal static one (since the latter is based on approximate information to obtain an "optimal" schedule) [Polychronopolus88].

3.5 Task System

A task system can be represented by a graph $G(V,E)$, where the number of tasks in the system is represented by $n = |V|$ and $|E|$ represents the number of edges [Samadzadeh92a]. Depending on the number of edges in the graph G being zero or non-zero, task systems are classified as either independent or dependent task systems.

A task system is said to be independent if $|E| = 0$. That is, none of the tasks in the task system communicate with any other task. If this task system represents a real program, we can say that there is no control dependency or data dependency among the components in the program. Since there is no inter-task dependencies, scheduling of independent tasks can occur in any order. Besides, no special considerations are necessary to determine the particular processor on which a given tasks must be scheduled in order to avoid or minimize the communication overhead. Such a concern is necessary for scheduling of tasks that are dependent on one another [Samadzadeh92a].

On the other hand, if $|E| > 0$, the task system is said to be dependant. Dependent task system graphs are assumed to be precedence graph. Because of the dependency constraints imposed on task system graphs through the existence of arcs, the scheduling mechanism must schedule these tasks in such a way that the correct execution sequence of the program is guaranteed [Samadzadeh92a].

CHAPTER IV

STATIC SCHEDULING OF INDEPENDENT TASKS

This chapter gives the description of an experiment performed [Thobani92] to investigate the static scheduling of independent tasks at the user level on the shared-memory multiprocessor Sequent Symmetry S/81 system running the DYNIX/ptx operating system.

4.1 Experimentation Platform

This section presents a brief overview of the computer system on which the preliminary experiment was performed.

The Sequent Symmetry systems offer four levels of computing capacity. The machine used for the preliminary experiment was a Symmetry S/81. The S/81 is Sequent's most powerful mainframe-class multiprocessor computer system featuring up to thirty Intel 80386 microprocessors (the Sequent computer in the Computer Science Department has 24 processors) operating at 20 Mhz, each with 128 kilobytes of cache memory. In its present configuration, the system has 384 megabytes of RAM, 43.3 gigabytes of hard disk storage, 256 directly-connected serial ports, four Ethernet ports, eight high-speed synchronous communication ports, eight parallel printer ports, and four 1/2-inch reel-to-reel tape drives [SEQ90]. The new product line of S/81 uses 80486 microprocessors instead of 80386s.

The Sequent Symmetry S/81 system runs the DYNIX/ptx operating system. DYNIX/ptx is a complete UNIX system port that is compatible with

AT&T System V 3.2. The extensive base of UNIX-compatible software can run on Symmetry systems with little or no change [DYNIX90d]. Sequent's operating systems have been engineered to use the parallel processing capabilities of Sequent computers [DYNIX90d].

4.2 Objectives of the Experiment

The objective of the preliminary experiment was to take n independent tasks, statically schedule them on p available processors, and then observe the termination sequence of the tasks as compared to the lengths of the tasks (i.e., the expected termination sequence of the tasks) for the purpose of finding out how the system actually behaves. The result of the preliminary investigation were intended to be put to use later in the scheduling of simulated dependent task systems, and eventually to be put to use in the scheduling of actual user tasks.

4.3 Details of the Experiment

In this experiment, 20 processors out of the 24 available in the Sequent Symmetry system were used. One hundred tasks were scheduled on 19 of the processors. The reason for using 19 processors out of 20 was that one of the processor was used as a master clock to help keep track of the progress of the tasks on the other 19 processors. The arrival and departure time of each task was recorded by this master clock.

4.3.1 Description of the Tasks

Two categories of tasks were used in the preliminary experiment. The tasks in the first category were of type "user program" because they contained no system calls. The tasks in the second category contained

operating system calls. For this experiment, the tasks were not obtained from actual users, instead they were simulated within the scheduler. This was done because of several reasons: actual historical data is not generally readily available in consistent and reliable form, contrived tasks can be targeted and fine-tuned essentially arbitrarily, and true "controlled" experiments that exclude extraneous conditions are only possible with contrived tasks especially on a multiprocessor in a multiuser and multi-programmed environment.

Therefore, there were basically only two unique tasks. By changing the lengths of the two tasks (i.e., processing time required), "different" tasks were simulated. Four sets of tasks, each containing 100 tasks, were created for each category. Hence, altogether there were 8 sets containing 100 tasks each. The experiment was performed on each of these sets. Graphs were plotted for the termination sequence of the tasks, Gantt charts of the processors involved were drawn, and finally the progress charts of the tasks were graphed (APPENDIX B).

4.3.2 Task Category 1 (User Program)

To simulate a task, which represents a user program or a user task, a while loop was used. The control variable of the while loop represents the length of the task. The source code of a typical "user program" is given below.

```
val = 0;
while (val <= length * WHILE_CT)
    val++;
```

In this program, the value of WHILE_CT was set to 800,000. Hence, when the value of length was 1 unit, the while loop executed 800,000 times. It was found that it takes approximately 1 second to execute this while loop 800,000

times. Using the above program segment, tasks representing user programs of different processing lengths were simulated by assigning different values to the while loop control variable.

4.3.3 Task Category 2 (Including a System Command)

The system command **sleep** was used in the second category of tasks. **sleep** delays the execution of a given command for a specified time interval (in seconds). The syntax of the **sleep** command is as follows.

sleep length command

In this system call, length is the time in seconds for which the processing of the command is to be delayed. For example, to delay the execution of the command `du -s /t/userid` for 5 seconds, the format of **sleep** command is as follows.

sleep 5 du -s /t/userid

Using the above system call, tasks that included system commands of different processing lengths were simulated by assigning different values to the length field of the **sleep** command.

It was found during the experiment that for length = 100, on the average the user program (i.e., the while loop) took 100.8 seconds whereas the system command (i.e., **sleep**) took 100 seconds. Hence, one can say that the sizes of the tasks in both categories for a particular value of length are comparable.

4.3.4 Description of the Eight Sets of Tasks

Having described the two distinct basic tasks and how the different tasks in each category were obtained using the basic tasks, we now discuss how the eight sets of tasks were simulated in this experiment.

Task Set One The first set of 100 tasks was simulated using the basic task of a user program (i.e., the while loop) and increasing the value of length (as discussed in Section 4.3.2) from 1 to 100. In this set of 100 tasks, the length of a task was obtained from the following formula.

$$\text{length of task} = \text{Task ID}$$

Therefore, a task whose Task ID is 10 has length equal to 10 and a task whose Task ID is 80 has length equal to 80. This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts in APPENDIX B, this set is represented by the following quadruple.

(19, 100, WHILE, INCR)

where

19 = number of processors available

100 = number of tasks

WHILE = user program (while loop)

INCR = task length is increasing with the Task ID

Task Set Two The second set of 100 tasks was simulated using the basic task of a user program (i.e., the while loop) and decreasing the value of length (as discussed in Section 4.3.2) from 100 to 1. In this set of 100 tasks, the length of a task was obtained from the following formula.

$$\text{length of task} = 100 - \text{Task ID}$$

Therefore, for instance a task whose Task ID is 10 has length equal to 90. This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, WHILE, DECR)

Task Set Three The third set of 100 tasks was simulated using the basic task of a user program (i.e., the while loop) and assigning the same length to all tasks. The value of length (as discussed in Section 4.3.2) was 100. This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, WHILE, EQUAL)

Task Set Four The fourth set of 100 tasks was simulated using the basic task of a user program (i.e., the while loop) and assigning random lengths (as discussed in Section 4.3.2) between 100 and 1. In this set of 100 tasks, the length of a task was obtained from the following formula.

length of task = 100 * rand_gene()

The rand_gene() routine returns a value between 1 and 0. This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, WHILE, RAND)

Task Set Five The fifth set of 100 tasks was simulated using the basic task containing a system command (i.e., the sleep command) and assigning increasing values for the length (as discussed in Section 4.3.3) from 1 to 100. In this set of 100 tasks, the length of a task was obtained from the following formula.

length of task = Task ID

This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, SLEEP, INCR)

Task Set Six The sixth set of 100 tasks was simulated using the basic task containing a system command (i.e., the **sleep** command) and assigning decreasing values for the length (as discussed in Section 4.3.3) from 100 to 1. In this set of 100 tasks, the length of a task was obtained from the following formula.

$$\text{length of task} = 100 - \text{Task ID}$$

This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, SLEEP, DECR)

Task Set Seven The seventh set of 100 tasks was simulated using the basic task containing a system command (i.e., the **sleep** command) and assigning equal lengths to all tasks, the value of length (as discussed in Section 4.3.3) was 100. This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, SLEEP, EQUAL)

Task Set Eight The eighth set of 100 tasks was simulated using the basic task containing a system command (i.e., the **sleep** command) and assigning random lengths (as discussed in Section 4.3.3) between 100 and 1. In this set of 100 tasks, the length of a task was obtained from the following formula.

$$\text{length of task} = 100 * \text{rand_gene}()$$

The `read_gene()` routine returns a value between 1 and 0. This set of 100 tasks was then submitted to the static scheduler that scheduled them onto the 19 available processors. In the graphs and charts given in APPENDIX B, this set is represented by the following quadruple.

(19, 100, SLEEP, RAND)

4.3.5 Static Scheduling of Tasks

Static scheduling of the 100 tasks to the 19 processors was done using the DYNIX/ptx parallel library routine `m_fork()`. Each task was assigned a Task ID and the scheduling was done in the ascending order of the Task IDs. The static scheduling of the 100 tasks onto the 19 available processors is given in TABLE I.

4.4 Analysis of the Results

The data collected as a result of this initial experiment includes the termination sequence of the tasks, the time at which a task was assigned to a processor, and the time at which a task left the scheduler (i.e., processing was completed). It should be noted that the arrival time of all tasks was taken to be zero, as all the tasks in a set were submitted to the scheduler at the same time.

Three graphs were plotted for each task set (see APPENDIX B). The first set of graphs (TERMINATION SEQUENCE) was plotted with task length vs. termination sequence. They show the effect of task lengths on the termination sequence. The second set of graphs (GANTT CHART) depict the Gantt charts of the 19 processors. For each processor, they show the time at which a processor started processing a task and the time at which the processing of that particular task was completed. Gantt charts are helpful in

finding individual as well as overall idle times for processors. The third set of graphs (PROGRESS CHART) plots Task ID vs. master clock. These graphs give a good picture of the amount of time that the tasks spent waiting in the system, the time a task was assigned to a processor, and the time at which a task left the system.

TABLE I

STATIC SCHEDULING OF 100 TASKS ON 19 PROCESSORS

Processor	Tasks Assigned (Task ID)					
0	0	19	38	57	76	95
1	1	20	39	58	77	96
2	2	21	40	57	78	97
3	3	22	41	58	79	98
4	4	23	42	59	80	99
5	5	24	43	60	81	-
6	6	25	44	61	82	-
7	7	26	45	62	83	-
8	8	27	46	63	84	-
9	9	28	47	64	85	-
10	10	29	48	65	86	-
11	11	30	49	66	87	-
12	12	31	50	67	88	-
13	13	32	51	68	89	-
14	14	33	52	69	90	-
15	15	34	53	70	91	-
16	16	35	54	71	92	-
17	17	36	55	72	93	-
18	18	37	56	73	94	-

4.4.1 Task Set One (19, 100, WHILE, INCR)

In this set, each task length was equal to the Task ID and, since the tasks were scheduled in the ascending order of the Task IDs, a straight line was expected in Figure 1.a which shows the termination sequence. It could be seen that at four places (marked with an * in Figure 1.a), a task with greater processing length terminated before a task with a comparatively smaller

processing length, although both tasks were scheduled at the same time. This apparent conflict is marked with an * in the Gantt chart (Figure 1.b) and in the corresponding progress chart (Figure 1.c).

One obvious reason for this apparent conflict seems to be the availability of processors to the scheduler. Since the scheduler scheduling the contrived tasks is running on top of the operating system, it is possible that when the scheduler asks the operating system for a particular processor, if that processor happens to be busy at the time, the operating system would dispatch the requested processor to the scheduler a short interval of time later than if the processor does not happen to be busy. Therefore, it could be argued that a smaller task running on a busier processor may take longer to process, in terms of the turnaround time, than a comparatively larger task running on a less busy processor.

4.4.2 Task Set Two (19, 100, WHILE, DECR)

If we look at the termination sequence for this task set (Figure 2.a), it can be seen that the first nineteen terminated tasks have task lengths between 81 and 99 (i.e., the length of the task with Task ID 1 is 99, the length of the task with Task ID 2 is 98, and so on). Within this range, the tasks seem to be terminating in the expected ascending order of their lengths, except a few cases where there seems to be a discrepancy (discussed in the analysis of task set one in Subsection 4.4.1). These apparent conflicts are marked by an * in Figure 2.a). Figures 2.b and 2.c give the Gantt chart and the progress chart for this set of tasks. In both of these figures, the apparent conflicts are marked with an * for the first 19 scheduled tasks (i.e., Task IDs 1 through 19).

4.4.3 Task Set Three (19, 100, WHILE, EQUAL)

For this set of tasks, the termination sequence (Figure 3.a) doesn't give much useful information. Ideally, all tasks that were scheduled at the same time should terminate at the same time. Looking at the Gantt chart (Figure 3.b), it can be seen that this is indeed what occurred most of the time, except some cases where a task terminated before another, not in conformity with expectation. This apparent conflict could be seen on both figures. For convenience a few of these apparent conflicts are marked with an * in Figure 3.b, the Gantt chart and in Figure 3.c, the progress chart. The justification for this apparent conflict is similar to the justification given in the discussion of task set one.

4.4.4 Task Set Four (19, 100, WHILE, RAND)

For this set of tasks, the termination sequence (Figure 4.a) doesn't give any useful information. Based on the Gantt chart (Figure 4.b), it can be seen that, among the first 19 tasks, the one with the shortest length (marked with an *) terminated first. From the progress chart (Figure 4.c), it can be seen that the first 19 tasks are terminating in the ascending order of their lengths as expected, except for some apparent conflicts. The justification for the apparent discrepancy is similar to the situation in task set one.

4.4.5 Task Set Five (19, 100, SLEEP, INCR)

In this task set, each task length was equal to the Task ID, and since the tasks were scheduled in the ascending order of Task IDs, a straight line was expected in Figure 5.a for the termination sequence. It could be seen that, at several places, a task with a greater processing length terminated before a task with a comparatively smaller processing length, although both

tasks were scheduled at the same time. A few of these apparent conflicts are marked with an *. These cases are also marked with an * in the corresponding Gantt chart (Figure 5.b) and progress chart (Figure 5.c). The justification for these apparent conflicts is similar to the situation in task set one.

4.4.6 Task Set Six (19, 100, SLEEP, DECR)

If we look at the termination sequence (Figure 6.a) for this task set, it can be seen that the first nineteen terminated tasks have task lengths between 81 and 99 (i.e., the task with Task ID 1 has length 99, the task with Task ID 2 has length 98, and so on). Within this range, the tasks seem to be terminating in the expected ascending order of their lengths, except for some apparent conflicts (marked with an *). For the first 19 scheduled tasks (i.e., Task IDs 1 to 19) these apparent discrepancies are also marked with an * in the corresponding Gantt chart (Figure 6.b) and progress chart (Figure 6.c). The justification for these apparent conflicts is similar to the situation in task set one.

4.4.7 Task Set Seven (19, 100, SLEEP, EQUAL)

For this set of tasks, the termination sequence (Figure 7.a) doesn't give much useful information. Ideally, all tasks that were scheduled at the same time should terminate at the same time. Looking at the Gantt chart (Figure 7.b), it can be seen that this is indeed what occurred for most of the cases except for some cases where there seem to be some apparent conflicts (the justification for such apparent discrepancies was discussed in the analysis of task set one). For convenience, a few of these apparent conflicts are marked with an * in the Gantt chart (Figure 7.b) and progress chart (Figure 7.c).

4.4.8 Task Set Eight (19, 100, SLEEP, RAND)

For this set of tasks, the termination sequence (Figure 8.a) doesn't give any useful information. Based on the Gantt chart (Figure 8.b), it can be observed that, among the first 19 tasks, the one with the shortest length (marked with an *) terminated first. Figure 8.c shows the progress chart. It could be seen that for the first 19 tasks, the tasks are terminating in the ascending order of their lengths. There seem to be some apparent conflicts whose justification is similar to the situation in task set one.

4.5 Summary of the Preliminary Results

The results of static scheduling of the two categories of tasks were as expected, i.e., if scheduled at the same time, a task with a smaller processing length should terminate earlier than the one with a comparatively larger processing length. There were a few apparent deviations. For example, if scheduled at the same time, a task with a larger processing length could terminate before another task with a smaller processing length. The justification for this apparent conflict was discussed in the analysis of task set one (Subsection 4.4.1).

It was also found that there were more apparent discrepancies in category 2 than in category 1. The reason for this difference in the number of apparent conflicts is as follows. The user programs are cpu-oriented, i.e., the while loops keep the cpu busy until a higher priority job arrives or until the while count is over. The system commands, on the other hand, are not cpu-oriented because tasks that are **sleeping** are swapped out by the operating system and hence they wait passively for the duration of their sleep. Thus the **sleep** command lets the cpu be used by other processes, and later uses the

cpu when the sleep time interval is over. This discrepancy can also be justified by the fact that the operating system places the swapped process back in the ready queue at an undetermined time after the sleep period is over.

CHAPTER V

HEURISTIC SCHEDULING

This chapter discusses the design methodology and the results of the empirical study conducted for the performance evaluation of several scheduling algorithms on Sequent Symmetry S/81 system. As discussed in Chapter III, the key performance measures used for this study are the schedule length and the performance ratio defined as the ratio between the schedule length on Sequent to the schedule length obtained from the different algorithms used for devising a static schedule for each task system. This chapter first discusses the scheduling of independent and dependent task systems in general, followed by a brief discussion of the four algorithms, namely, Variant-Load, Divide and Fold, Ranked Weight, and ESP/VL used for the scheduling of independent and dependent task system in this thesis. This chapter concludes with a discussion of the design methodology, the instrumentation, and the results obtained in this study.

5.1 Notation and Definitions

In this section, the notation and definitions used in the discussions of this chapter are introduced. A task system is represented as a graph $G(V,E)$ where V represents the set of nodes in a task system and E represents the dependencies between pairs of tasks in a task system. In the case of independent task systems, the edge set E is empty. Associated with each task is a non-negative integer that represents the estimated processing time of the

task. The sequential execution time of a task system is defined to be the sum of the processing times of all of the tasks in a task system, and is denoted by T_s . We denote the parallel execution time or the schedule length of a task system by T_p . Determining an exact value for T_p is not possible because it depends on many factors such as the scheduling algorithm used, the multiprocessor environment employed, and the number of processors used for the parallel execution of a task system.

We are interested in the speed-up of execution of a task system T which executes on p processors and produces a parallel execution length T_p . The speed-up of a task system T is denoted by S_T , and it is calculated using the formula

$$S_T = T_s / T_p$$

For the purpose of the current study, T_p is taken to be the schedule length that is obtained by executing the task system partitions created by the scheduling algorithms used in this thesis on Sequent. Therefore, T_p includes the scheduling overhead involved in run-time monitoring of the tasks being executed. Normally, the speed up of execution is calculated in order to measure the relative merits of a parallel algorithm. Unlike the above definition, the purpose of calculating the speed-up in the current study is to measure and evaluate the system overhead involved in the scheduling of task systems. Therefore, we will compare S_T with the speed-up of the execution when run-time overhead is ignored. This speed-up is denoted by S_{max} and is defined as

$$S_{max} = T_s / T_A$$

where T_A is the schedule length produced by one of the four scheduling algorithms used in this study on p processors. Since it is expected that $S_T \geq S_{max}$, the difference $S_T - S_{max}$ is defined to be the slowing-down factor

incurred in run-time monitoring of the self-timed scheduling approach used in this study.

As discussed earlier, we are interested in the performance ratio of the execution of task systems on Sequent. We denote the performance ratio of the execution of a task system on p processors as $R_p(A)$, and define it as follows

$$R_p(A) = T_p / T_A$$

where A is the schedule (i.e., the static task partitions) produced by one of the four scheduling algorithms used in this study, T_p is the schedule length of schedule A on Sequent, and T_A is the estimated length of schedule A on p processors produced by one of the four scheduling algorithms. Notice that T_A does not include any run-time overhead. We would like $R_p(A)$ to be as close to 1 as possible. The performance ratio used in this study is a measure that indicates the relative performance of a schedule on a given number of processors, and shows the run-time overhead involved in interprocessor synchronization using p processors.

In order to determine the efficiency with which the processors are utilized, we define another measure, referred to as E , and define it as the ratio of the speed-up to the number of processors used

$$E = S_T / p$$

Efficiency is a measure that is related to resource utilization, while speed-up is related to the schedule length for a given task system. Later in this chapter, we will observe that E and S_T are related to conflicting goals and therefore, it might not be possible to maximize both.

5.2 Scheduling of Independent Task Systems

A task system can be represented as a graph $G(V,E)$, where V represents the set of nodes and E represents the set of edges in the graph. In

the graphic representation of a task system, the nodes represent the tasks and the edges represent the dependencies between tasks. The number of tasks in the system is defined as $n = |V|$ and the number of dependencies in the task system is defined as $|E|$.

For an independent task system, the number of edges $|E|$ is zero. There are no communication or data dependencies among the tasks in an independent task system, hence tasks could be scheduled in any order on the available processors and no special consideration is needed to assign a particular task to a particular processor in order to minimize the communication overhead.

The main objective of the scheduling of independent tasks is to balance the workload among the available processors. In the ideal case, all the processors should complete the execution of their assigned tasks at the same time [Samadzadeh 92c]. The next two subsections describe the independent task system scheduling algorithms studied in this thesis.

5.2.1 Variant-Load Algorithm

The Variant-Load algorithm, developed by Samadzadeh [Samadzadeh 92c], is designed for the scheduling of independent tasks on a given number of processors p , with the objective of balancing the assigned workload on all processors. This algorithm is developed based on concepts from *bin packing* [Coffman78][Johnson74].

Bin packing is a general scheme in which n items of different sizes must be packed into k bins of capacity C each, where the sum of the sizes of the n items is in general less than or equal to $k*C$. The Variant-Load algorithm treats each available processor as a bin with a certain capacity

(workload capacity) that is filled (packed) with variable-sized processing times associated with the tasks [Samadzadeh92a].

5.2.2 Divide and Fold Algorithm

Divide and Fold Algorithm (D&F) developed by Polychronopoulos [Polychronopoulos86], tries to devise a schedule with the shortest possible schedule length, given a set of independent tasks and a number of processors. The D&F algorithm operates in two phases. It starts with a list that is sorted in non-ascending order based on the task processing times. During phase I, it repeatedly divides the sorted list in half and folds the two halves into one list. Assuming that the list starts with n partitions of one element each, dividing and folding is performed on the list such that after the first iteration there are $n/2$ partition, and each partition has 2 elements. The dividing and folding of the list continues until there are as many partitions as the number of available processors, and each partition contains n/p elements. In the second phase, three tests are performed to further balance the workload among the processors. In this phase, the workloads between two pairs of partitions are compared and three tests are performed. Depending on the results of these tests, the elements (i.e., tasks) are moved among the partitions such that their shifting would further balance the workload assigned to the processors.

5.3 Scheduling of Dependent Task Systems

For a dependent task system, the number of edges, $|E|$, in the task system graph $G(V,E)$ is greater than zero. Dependent task systems contain communication or data dependencies among the tasks, hence the tasks cannot be scheduled in any order on the available processors. As a result,

special considerations are necessary for the assignment of a particular task to a particular processor in order to generate a correct execution sequence. The main objective of the scheduling of dependent task system is the same as that of independent task systems (i.e., to balance the workload among the available processors). In the ideal case, all the processors should complete the execution of their assigned tasks at the same time [Samadzadeh92c]. However, because of the inter-task dependencies and communication delays, such a goal may not be always achievable.

5.3.1 Ranked Weight Algorithm

The Ranked Weight algorithm, used for the scheduling of dependent task systems, was developed by Samadzadeh [Samadzadeh 92b, 92c]. This algorithm operates by initially dividing the tasks in a task system into a number of independent layers. The algorithm later selects a task that is at the highest level, has the largest ranked weight, and all of whose predecessors have finished execution. The ranked weight of a task in the task system is equal to the processing time required by that task and all of its successors. For example, the ranked weight of the source node is equal to the total execution time of the entire task system.

The implicit priority rules embedded in the Ranked Weight algorithm are as follows. By selecting tasks that are at the highest level, this algorithm takes advantage of ESP partitioning (Earliest Schedule Partitioning scheme) and schedules the tasks that are ready at the earliest possible time [Samadzadeh 92c]. By using the ranked weights of tasks, this approach implicitly incorporates several different priorities. The algorithm gives priority to a) tasks with long execution times, b) tasks with the largest

number of immediate successors, and c) tasks with successors that have long processing times.

The Ranked Weight algorithm produces a static assignment of tasks to processors. As the static assignment of tasks to processors is created, the Ranked Weight algorithm also keeps track of the idle times that might be introduced into the schedule, in order to satisfy the timing and precedence constraints for the scheduling of the tasks. The output from the Ranked Weight algorithm consists of the partitioning of the tasks into p ordered sets, and an estimate of the schedule length for the execution of the given task system on p processors.

5.3.2 ESP/VL Algorithm

The ESP/VL algorithm for scheduling of dependent task systems, developed by Samadzadeh [Samadzadeh92c], uses the ESP algorithm and the Variant-Load algorithm [Samadzadeh92b, 92c]. Given a task system, the ESP algorithm divides the task system into a number of layers such that the tasks in each layer are independent of one another. The Variant-Load algorithm is a near-optimal algorithm that can be used for the scheduling of independent tasks. The ESP/VL algorithm uses the partitions created by the ESP algorithm and schedules the tasks in the resulting layers using the Variant-Load algorithm. Because of the communication and data dependencies existing among the tasks, the execution of the tasks in a subsequent layer cannot start until all the tasks in the previous layer have finished execution. The schedule length for the ESP/VL algorithm is defined to be the sum of the schedule lengths for each independent layer scheduled by the Variant-Load algorithm.

5.4 Task System Generator

In this thesis, task systems are represented as random graphs of varying topologies. The task system generator used in this study takes as input the number of tasks in the desired task system and a range of processing times in the case of independent task systems. The output of the task system generator is the desired number of tasks with the specified range of processing times.

In the case of dependent task systems, the input expected consists of the number of tasks, their expected range of processing times, and the probability for the existence of the precedence relations among the tasks. The precedence probability is a number in the range $0 < P \leq 1$.

With smaller values of P there are fewer precedence constraints, and thus the graph generally exhibits a higher degree of parallelism. The output from the task generator algorithm consists of an upper-triangular adjacency matrix of zeros and ones, in which a 1 indicates that the tasks at the intersecting row and column are inter-dependent. It is easy to see that the 1 entries in each row constitute the successors of the task indexed by the given row, and the 1 entries in each column are the predecessors of the task indexed by a given column. The task system adjacency matrix is used later to identify the tasks ready for scheduling on Sequent.

5.5 Scheduling Methodology

The main objective of the second part of this thesis was evaluation of the performance of schedules produced by the four algorithms described in Sections 5.2 and 5.3 on Sequent using a self-timed scheduling approach.

We were interested in evaluating the overhead involved in the scheduling of tasks in dependent and independent task systems on Sequent.

The tasks scheduled in this empirical study were contrived tasks that do not incur any input/output overhead. The rationale for the choice of contrived tasks as opposed to real user code was that, at that stage, we were primarily interested in measuring the task start-up time and processor synchronization overhead of the scheduling approaches employed in this study. Therefore, the use of the contrived tasks that involve a fixed amount of processing time would help us control the dependent variables used in this study more accurately.

5.5.1 Self-Timed Scheduler

The scheduling algorithms studied in this thesis produce both a static assignment of the tasks in a task system to processors, and also an estimate of the schedule length for the given number of processors. In order to provide run-time support for the execution of the scheduled tasks, a scheduler was developed that assigned tasks to physical processors and monitored their performance on Sequent.

The task system scheduler developed in this study uses a self-timed scheduling approach. In a self-timed scheduling approach, the tasks in a task system are partitioned into a number of ordered sets on a static basis. Once the execution of the task system begins, each partitioned task set is assigned to a physical processor. After the initial assignment of task sets to processors, the responsibility of determining the start time of the tasks rests with the processor to which a task is assigned.

As described earlier, the scheduling algorithms described in Sections 5.2 and 5.3 produced a static schedule for the given task systems. These static schedules were made available to physical processor by the self-timed scheduler used in this study. The self-timed scheduler operates by obtaining

an appropriate number of processors and creating a task queue for each of the processors. The tasks assigned to each processor are placed in that processor's queue. The tasks in each queue are processed on a FIFO basis.

For independent task systems, each of the processors operate independently of one another and no synchronization is necessary. However, because of inter-task dependencies, in the case of dependent task systems, the processors must synchronize the timing of the execution of their assigned tasks. Under the self-timed scheduling approach described in this subsection, the adjacency matrix corresponding to a dependent task system was used as a shared data structure that monitors the execution of a task system. Before each processor starts the execution of the task at the head of its FIFO task queue, it checks the list of predecessors of that task (i.e., the corresponding column in the adjacency matrix). If the processor finds that the predecessor list corresponding to a given task is empty (i.e., all entries in that column of the matrix are zeros), the processor starts the execution of the task. If the task at the head of the processor queue is not ready, the processor repeatedly checks the adjacency matrix. After a ready task is executed, the processor assigned to the task changes all the 1 entries in the list of successors of the executed task to 0's (by obtaining an exclusive lock for the adjacency matrix), to signal the completion of this task to other processors.

5.5.2 Instrumentation

This empirical study was performed on the Sequent Symmetry S/81 computer. The S/81 is Sequent's most powerful mainframe-class multiprocessor computer system featuring up to thirty Intel 80386 microprocessors operating at 20 MHZ, each with 128-kilobytes of cache

memory. In its present configuration, the system has 384 megabytes of RAM and 43.3 gigabytes of hard disk storage [SEQ90].

In order to get accurate results, a benchmarking program (called TEAM) that has been developed for running in single user mode, for BSD-based UNIX, was used. TEAM is a program available through *netlib*. Sequent S/81 runs under the DYNIX/ptx operating system. DYNIX is a SystemV-based environment. The TEAM program obtained through *netlib* was modified to run on DYNIX. The operating system on Sequent gives the highest priority to a program running under TEAM. TEAM also disables storage swapping, page fault frequency adjustment, and process aging, hence a task running under TEAM will incur minimal overhead [Samadzadeh 93].

The empirical study, that is the second part of this thesis, was aimed at measuring the scheduling and processor synchronization overhead for a self-timed scheduling approach in a multiprogrammed, time-shared environment (i.e., the Sequent multiprocessor). Because of the possible interference from other user jobs and the operating system overhead involved (e.g., paging and quantum allocation), using the TEAM program allowed us to monopolize the required number of processors for the duration of the task execution times. We were also able to disable storage swapping and process aging in order to execute the task systems with minimal system overhead.

5.6 Task System Characteristics Used in the Simulation

As discussed in Section 5.4, task systems were generated for both independent and dependent task sets. For independent task systems, the control variables were the number of tasks and the range of the processing times of the tasks; whereas for dependent task systems, in addition to the above two variables, the degree of parallelism was also used as a control

variable. The following subsections discuss the characteristics of the independent and dependent task systems used in this study.

5.6.1 Independent Task Systems

Two tests were performed for dealing with the scheduling of independent task systems. The control variables used in these tests were the number of tasks in a task system, the processing times of the tasks, and the number of processors used for the scheduling of the tasks. The number of tasks used for this test was $n = 100$ (i.e., the number of tasks was kept constant) and the dependent variables were task processing times (for this test, processing times were in the range of 1 to 5) and the number of processors p (for this test, the number of processors were increased from 2 to 8, with increments of 1).

In the second test, the number of processors was kept constant to eight (i.e., $p = 8$), and the dependent variables were the task processing times (in the range of 1 to 5) and the number of tasks n (for this test, the number of tasks was increased from 50 to 100 with increments of 10). The reason for changing the values of some of the variable was to see whether the performance ratio would be affected with different variables changing in a "controlled" manner.

5.6.2 Dependent Task Systems

The test that was performed for scheduling of dependent task systems is described in this subsection. The control variables used in this test were the number of tasks in the task system, the processing times of the tasks, and the number of processors used for the scheduling of the tasks. In this test, the number of tasks, was kept constant (i.e., $n = 50$). The dependent variables

were the number of processors p (increased from 2 to 8 with increments of 1) and the processing times of the tasks which were in the range of 1 to 8.

Randomly generated task systems were created using the task system generator described in Section 5.4. The probability for the existence of inter-task dependencies was set to $P = 0.1$. This value was selected based on our earlier experiments with properties of task systems under different probabilities [Samadzadeh 93]. Based on a sample of 500 task systems generated using a probability of 0.1, the degree of parallelism exhibited in the graphs is in the range of 8 to 16 concurrent tasks. For smaller values of P , the degree of parallelism increases. Obviously, concurrency decreases for larger values of P .

5.7 Results of the Experiment

The purpose of this experiment was to measure the behavior of Sequent under a run-time support system for the execution of task systems. The scheduling algorithms used in this study created a static assignment of tasks to processors. Therefore, run-time support was needed to synchronize the timing of the execution of different tasks. In order to guarantee the correct execution of the tasks in a dependent task system, the run-time support had to monitor and detect the completion times of the predecessors of a given task before it could release that task for execution. Of course, it is evident that determining the timing of execution of tasks applies to the execution of tasks in dependent task systems only.

The results of the experiments are summarized in terms of the parameters defined in the Notation and Definition section of this chapter (Section 5.1). In TABLES II through VII, n represents the number of nodes in a task system, p represents the number of processors used to execute the task

system, R is the performance ratio, E is the efficiency, S is the speed-up of execution of the task system using p processors, S_{\max} is the speed-up of execution without the inter-processor synchronization overhead of Sequent, and $S_{\max} - S$ represents the overhead incurred in the scheduling of the task systems on Sequent. The parameters R , E , S , and S_{\max} are defined in Section 5.1.

5.7.1 Execution of Independent Task Systems

The characteristics of the task systems used in the scheduling of independent task systems was described in Section 5.6.1. The results of these tests are reported in TABLES II through V. Each row in these tables represents the average for a total of twenty different task systems scheduled. For example, in TABLE II, twenty independent task systems, consisting of 100 tasks each, were generated and each of the twenty task systems were scheduled using 2 to 8 processors. Therefore, the results presented in TABLE II are the summary of 140 different schedules executed.

The scheduling algorithms used for the scheduling of independent tasks on Sequent were Variant-Load and D&F (discussed in Section 5.2). Once static schedules were created using the Variant-Load and D&F algorithms, the partitioned task sets were placed in queues associated with the physical processors employed for the execution of the task sets. Because of the nature of processing, no interprocessor communication was necessary and therefore, the processors execute the tasks assigned to them on a FIFO basis as they picked up tasks from the head of their associated queue.

As reported in Section 5.6.1, two different tests were performed for scheduling of the independent task systems. The results shown in TABLES II and IV relate to the first test where the number of tasks in a task system was

kept constant for each of the two independent task system schedules used in this study, while the number of processors used was increased from 2 to 8 with increments of 1. TABLES III and V show the results of the second test in which the number of processors used was kept constant while the number of tasks in each task system was increased from 50 to 100 with increments of 10.

Recall that the definition of performance ratio, as given in Section 5.1, is the ratio of the obtained schedule length on Sequent to the schedule length produced by the scheduling algorithms used in this study. Because there is no interprocessor synchronization overhead involved in executing the independent task systems, it can be noticed in TABLES II, III, IV, and V that the performance ratio R is always either 1 or very close to 1. This observation indicates that the task start-up overhead on sequent is very small and therefore negligible. We will use this observation for drawing some conclusions for the scheduling of dependent task systems in the next section.

TABLE II

RESULTS OF TEST 1 FOR THE SCHEDULING OF AN INDEPENDENT TASK SYSTEMS USING THE VARIANT-LOAD ALGORITHM

n	p	R	E	S	S_{\max}	$S_{\max}-S$
100	2	1.00	1.00	2.00	2.00	0.00
100	3	1.00	0.93	2.98	2.99	0.01
100	4	1.10	1.00	4.00	4.47	0.47
100	5	1.01	0.80	4.90	4.95	0.05
100	6	1.00	0.99	5.96	5.96	0.00
100	7	1.00	0.99	6.91	6.91	0.00
100	8	1.01	1.00	8.00	8.02	0.02

In TABLES II through V, S_{\max} represents the maximum speed-up that can be realized in the execution of a given task system on p processors under the given scheduling algorithm. The definition of S_{\max} is given in Section 5.1. As described earlier, S is analogous to S_{\max} with the exception that it contains the task start-up overhead incurred by Sequent. Therefore, the last column in TABLES II through IV represents the amount of slow-down in the speed-up compared to the maximum speed-up exhibited by the schedule.

TABLE III

RESULTS OF TEST 2 FOR THE SCHEDULING OF AN INDEPENDENT TASK SYSTEMS USING THE VARIANT-LOAD ALGORITHM

n	p	R	E	S	S_{\max}	$S_{\max} - S$
50	8	1.00	0.98	7.84	7.84	0.00
60	8	1.01	0.98	7.87	7.91	0.04
70	8	1.00	0.99	7.96	7.76	0.00
80	8	1.00	0.99	7.96	7.96	0.00
90	8	1.00	0.96	7.88	7.88	0.00
100	8	1.00	1.00	8.00	8.00	0.00

The Efficiency E in TABLES II through IV relates to the processor utilization for the execution of task systems given p processors. It can be observed that, in the case of independent task systems, the utilization is very high.

The speed-up S in TABLES II and IV increases linearly as the number of processors allocated for the execution of a task system increases. This observation indicates firstly that the scheduling algorithms used in this study are suitably capable of balancing the workload among the processors used, and secondly that the Sequent's overhead incurred in task start-up

remains proportionally constant regardless of the number of processors allocated to a task system.

TABLE IV

RESULTS OF TEST 1 FOR THE SCHEDULING OF AN INDEPENDENT TASK SYSTEMS USING THE DIVIDE AND FOLD ALGORITHM

n	p	R	E	S	S_{\max}	$S_{\max}-S$
100	2	1.00	1.00	2.00	2.00	0.00
100	3	1.00	0.93	2.98	2.98	0.00
100	4	1.00	1.00	4.00	4.00	0.00
100	5	1.00	0.98	4.90	4.90	0.00
100	6	1.00	0.99	5.96	5.96	0.00
100	7	1.00	0.96	6.75	6.75	0.00
100	8	1.00	1.00	8.00	8.00	0.00

TABLE V

RESULTS OF TEST 2 FOR THE SCHEDULING OF AN INDEPENDENT TASK SYSTEMS USING THE DIVIDE AND FOLD ALGORITHM

n	p	R	E	S	S_{\max}	$S_{\max}-S$
50	8	1.00	0.93	7.45	7.45	0.00
60	8	1.00	0.94	7.54	7.54	0.00
70	8	1.00	0.96	7.68	7.68	0.00
80	8	1.00	0.99	7.97	7.97	0.00
90	8	1.00	0.93	7.45	7.45	0.00
100	8	1.00	1.00	8.00	8.00	0.00

5.7.2 Execution of Dependent Task Systems

The characteristics of the task systems used for the scheduling of dependent task systems was described in Section 5.6.2. The results of these tests are reported in TABLES VI and VII. Each row in these tables represents the average of a total of twenty different task systems scheduled. The results in TABLES VI and VII are based on twenty different task

systems of varying topologies that were generated containing 50 tasks each. Each of the twenty task systems were scheduled on 2 through 8 processors. Therefore, each table shows the results of 140 different schedules.

The scheduling algorithms used for the scheduling of dependent task systems on Sequent were the Ranked Weight and ESP/VL algorithms. These algorithms create a static assignment of tasks to processors by partitioning the tasks in a task system into a number of potentially concurrent task sets. Because of the communication and data dependencies between pairs of tasks, run-time monitoring of the execution of these task systems is necessary to determine the exact execution time of each task.

The self-timed scheduling approach used in this study creates a run-time queue for each of the physical processors. Before each processor could pick up the next task for execution, it has to check the task adjacency matrix in order to determine whether its predecessor(s) are finished. The amount of time required to actively check the status of the predecessors of a task at the head of a processor queue, in order to detect the ready tasks, constitutes the run-time overhead involved in self-timed scheduling of tasks in a task system.

As reported in Section 5.6.2, the number of tasks in the task systems generated for the scheduling of dependent task systems was kept constant while the number of processors used for the scheduling of these task systems increased from 2 to 8 with increments of one. Task systems of varying topologies and inter-task dependencies were used. The number of concurrently executable tasks was defined to be in the range of 8 to 16. The degree of concurrency was defined using a probability factor (for the existence of precedence relationships) between pairs of tasks when the task systems were generated.

The definition of the performance ratio R in Section 5.1 describes the overhead involved in the scheduling of a task system on Sequent. Referring to the performance ratio tabulated in TABLES II through V regarding the scheduling of independent task systems, it can be noticed that the performance ratio is either 1 or very close to 1 in almost all of the cases considered. This observation indicates that in the case of independent task systems, there is no interprocessor synchronization involved, therefore the small existing overhead, where present, is due to task start-up times. Referring to the performance ratio reported in TABLES VI and VII, it can be concluded that the overhead incurred is mainly due to interprocess communication and synchronization.

As mentioned before, in a dependent task system a processor cannot start execution of a task at the head of its queue unless all the predecessors of that given task have completed their execution. The processor assigned to a particular task determines the status of that task's predecessors by checking the task adjacency matrix, which is a shared data structure. Therefore, the overhead exhibited in the performance ratios reported in TABLES VI and VII involves the time it takes for the allocated processors to update the task adjacency matrix by obtaining an exclusive access right to it.

An interesting observation relates to the fact that the performance ratio in TABLES VI and VII degrades somewhat as the number of processors increases and improves again with a larger number of processors afterwards. For example, we have $R = 1.16$ with 2 processors, 1.38 with 5 processors, and 1.27 with 8 processors in TABLE VI. A similar effect can be observed in TABLE VII. The justification for this change is that the performance ratio is better at the two extremes of the number of allocated processors because with a smaller number of processors (for example, $p = 2$) there is not much

contention over the adjacency matrix, which is a shared data structure used for enforcing precedence relations in the execution of tasks, and thus the overhead is low. A similar explanation for a larger number of processors (for example, $p = 8$) is due to the fact that the concurrency exhibited by the task system is supported better (recall that the degree of concurrency exhibited by the task systems in this study is in the range of 8 to 16). That is, the task throughput is faster with a larger number of processor and therefore, processors will be less apt to have to wait on one another until the predecessors of a given task finish their execution.

TABLE VI

RESULTS OF THE TEST FOR THE SCHEDULING OF A DEPENDENT TASK SYSTEMS USING THE RANKED WEIGHT ALGORITHM

n	p	R	E	S	S_{\max}	$S_{\max} - S$
50	2	1.16	0.81	1.63	1.89	0.26
50	3	1.23	0.72	2.16	2.69	0.53
50	4	1.34	0.60	2.42	3.28	0.86
50	5	1.38	0.51	2.59	3.50	0.91
50	6	1.24	0.51	3.05	3.78	0.73
50	7	1.32	0.42	2.97	3.84	0.87
50	8	1.27	0.40	3.18	4.05	0.87

As described in the introduction of Section 5.7, columns with headers S and S_{\max} refer respectively to the speed-up gained in the execution of a task system on Sequent and the speed-up realized by the schedule devised for a task system (using one of the two scheduling algorithms used in this study). The difference $S_{\max} - S$ shows the relative slow-down in speed-up due to processor synchronization.

TABLE VII

RESULTS OF THE TEST FOR THE SCHEDULING OF A DEPENDENT TASK
SYSTEMS USING THE ESP/VL ALGORITHM

n	p	R	E	S	S_{\max}	$S_{\max}-S$
50	2	1.17	0.77	1.55	1.81	0.26
50	3	1.20	0.66	1.99	2.39	0.40
50	4	1.19	0.62	2.50	3.01	0.51
50	5	1.16	0.52	2.62	3.05	0.43
50	6	1.19	0.43	2.56	3.05	0.49
50	7	1.16	0.37	2.62	3.05	0.43
50	8	1.17	0.33	2.62	3.05	0.43

Another measure used in the analysis of the results in this thesis is the efficiency E which refers to processor utilization. As noticed in TABLES VI and VII, the efficiency decreases with the increased number of processors. This indicates that the rate of growth of the speed-up of execution is not linearly proportional to the increase in the number of processors. This effect can be related to such factors as the amount of parallelism present in an application, the schedule length, and the communication overheads involved. It should be mentioned that efficiency is a measure that is related to the resource utilization at the system level and is not directly related to the improvement of the performance of a single job. Speed-up and efficiency in this context relate to conflicting goals and may not be both maximizable at the same time. For example, if the goal is minimization of the schedule length regardless of resource utilization, then executing a task system on eight processor in order to obtain a 4-fold speed-up and an efficiency of 0.5 is preferred to executing a task system on two processors with a 2-fold speed-up and an efficiency of 0.9.

5.8 Summary and Conclusions

The main objective of the studies reported in this chapter was to measure the suitability of the self-timed scheduling approach on a shared-memory multiprocessor environment such as Sequent. In the self-timed scheduling approach, task systems are scheduled on a static basis by initially partitioning the task system into a number of task sets. The timing for the execution of each task is not specified at the time of partitioning. Each of the task sets are assigned to a physical processor once the execution of the task system begins. The self-timed scheduler monitors the progress of task executions at run-time.

In order to observe the performance of a self-timed scheduling approach on Sequent, four different scheduling algorithms were used (two dependent task system scheduling algorithms and two independent task system scheduling algorithms). These algorithms created static schedules for task systems and provided an estimate of the schedule lengths on the given number of processors. The static schedules were later mapped onto an appropriate number of physical processors on Sequent.

A major concern in any performance study in a multiprogrammed, time-shared environment such as Sequent involves the caveat that interference from other concurrent users of the system can potentially invalidate the results of the study. In order to remedy this problem, two different measures were taken. Firstly, the performance tests were performed using a benchmarking program (TEAM which is available through *netlib*). This tool allowed us to monopolize the required number of processors for the duration of the tests performed in this study and secondly, all the tests described in Section 5.6 were repeated a number of times with a slightly different variation for the purposes of validation. In this validation test, a

single task system for each of the tests described in Sections 5.6.1 and 5.6.2 was generated and was run twenty different times under different system work loads. For the validation test, a total of 140 schedules were repeated for each of the tests reported in TABLES II through VII. The results obtained from the validation test were consistent with the results reported in this chapter.

CHAPTER VI

SUMMARY AND FUTURE WORK

6.1 Introduction

The main objective of the research performed in this thesis was two-fold. During the first phase, this research was concerned with investigating the behavior of static user-level scheduling of independent tasks on Sequent. During the second phase of this research, the objective was to investigate the performance of Sequent under a run-time support system for the execution of dependent as well as independent task systems. This chapter summarizes the results of these investigations and discusses the future work based on the results obtained from the investigations carried out.

6.2 Summary

For the purpose of the first phase of this research, it was necessary to see how the system behaved when tasks were scheduled by the user instead of the operating system. Preliminary experiments were performed to investigate the behavior of the system for "user-level" scheduling. The objective of the preliminary experiment was to take n independent tasks and statically schedule them on p available processors, and then to observe the termination sequence of the tasks for the purpose of finding out how the system actually behaved. The result of static scheduling of the two categories of tasks were as expected. That is, if scheduled at the same time, a task with a smaller processing length terminated earlier than the one with a

comparatively larger processing length. There were a few apparent deviations. Occasionally, if scheduled at the same time, a task with a larger processing length terminated before another task with a smaller processing length. The justification for this apparent conflict was discussed in the analysis of task set one (Subsection 4.4.1).

It was also found that there were more apparent discrepancies in category 2 than in category 1 as discussed in Sections 4.3.2 and 4.3.3. The reason for this difference in the number of apparent conflicts is as follows. The user programs are cpu-oriented, i.e., the while loops keeps the cpu busy until a higher priority job arrives or the while count is over. The system commands, on the other hand, are not cpu-oriented because tasks that are **sleeping** are swapped out by the operating system and hence they wait passively for the duration of their sleep. Thus the **sleep** command lets the cpu be used by other processes and later uses the cpu when the **sleep** time interval is over. This discrepancy can also be justified by the fact that the operating system places the swapped process back in the ready queue at an undetermined time after the **sleep** period is over.

As described in the introduction section, one of the main objectives of this thesis during the second phase of the investigations was evaluation of the performance of the schedules produced by the four scheduling algorithms described in Sections 5.2 and 5.3 on Sequent, using a self-timed scheduling approach. In order to investigate the performance of this approach on a shared-memory machine, static schedules were created using each of the four scheduling algorithms used in this study. The schedules produced by these algorithms were mapped onto processors at run-time and the system overhead incurred as a result of the run-time monitoring of the task system execution was recorded and measured.

The collected results indicated that task initiation overhead on Sequent is very small (almost negligible), and therefore run-time scheduling of the tasks on Sequent seems to be preferred to statically scheduled tasks. The overhead involved in processor synchronization for the parallel execution of task systems is dependent on the synchronization method used by the self-timed scheduling approach. The performance of the self-timed approach developed in this thesis was reported in terms of the performance ratio and the speed-up gained in the parallel execution of the task systems. The results of these tests are shown in TABLES II through VII.

6.3 Future Work

In this empirical study, the tasks used were simulated in the scheduler in order to have total control over the process (see Section 4.3.1 for more detail). One extension for this thesis would be to use real tasks for studying the system behavior. Tasks in this study were cpu-oriented and there was no actual I/O, hence no I/O overhead was studied. For future work, the scheduler used in this study could be modified so that tasks that perform I/O could be studied in order to measure the I/O overhead incurred on Sequent.

At this point, the scheduler can create schedules for a given task system using four different scheduling algorithms, namely, Variant-Load, Divide and Fold, Ranked Weight, and ESP/VL. The scheduler used in this thesis could be modified so that other scheduling algorithms could be tested as well.

Another area of improvement which can demonstrate significant results is to do further investigations on improving the processor synchronization overhead under the self-timed scheduling approach used in this thesis. In order to do this, further research needs to be done on the

feasibility of introducing locks with finer granularity on the shared data structure (i.e., the adjacency matrix) used for inter-processor synchronization.

REFERENCES

- [Amdahl67] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Computer Science Conference*, Vol. 30, 1967.
- [Archibald86] J. Archibald and J. Baer "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer System*, Vol. 4 (November 1986) 273-298.
- [Baer73] J. L. Baer "A Survey of Some Theoretical Aspects of Multiprocessing," *ACM Computing Survey*, Vol. 6, No. 1 (March 1973) 31-80.
- [Coffman78] E. G. Coffman, Jr., M. R. Garey, and D.S. Johnson, "An Application of Bin-Packing to Multiprocessor scheduling," *SIAM Journal of Computing*, Vol. 7, No. 1, (February 1978) 1-17.
- [DYNIX90a] *DYNIX/ptx C and Language Tools*, Sequent Computer Systems, Inc. (1990).
- [DYNIX90b] *DYNIX/ptx Operations Guide: ptx/ADMIN*, Sequent Computer Systems, Inc. (1990).
- [DYNIX90c] *DYNIX/ptx Reference Manual Section 1 L-Z*, Sequent Computer Systems, Inc. (1990).
- [DYNIX90d] *DYNIX/ptx User's Guide*, Sequent Computer Systems, Inc. (1990).
- [Graham76] R. L. Graham, *Bounds on the Performance of Scheduling Algorithm, Computer and Job/Shop Scheduling Theory* (E. G. Coffman, Ed.) John Wiley, New York, NY (1976).
- [Graham69] R. L. Graham, "Bounds on Multiprocessor Timing Anomalies," *SIAM Journal of Applied Mathematics*, Vol. 17, No. 2 (March 1969) 416-429.
- [Ha91] Soonhoi Ha and Edward A. Lee "Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iteration," *IEEE Transactions on Computers*, Vol. 40, No. 11, (November 1991) 1225-1238.
- [Johnson74] D. S. Johnson, A. Demers, J. D. Ullman, M.R. Garey, and R.L. Graham "Worst-Case Performance Bound for Simple One-Dimensional Packing Algorithm," *SIAM Journal of Computing*, Vol. 3, No. 4, (December 1974) 299-334.

- [Kwan90] Andrew W. Kwan, Lubomir Bic, and Daniel D. Gajski "Improving Parallel Program Performance the Using Critical Path Algorithm," *Languages and Compilers for Parallel Computing (Selected papers of the second workshop)*, Pitman Publishing, London, UK, 1990.
- [Lewis92] Ted G. Lewis, and Hesham El-Rewini, *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ, 1992.
- [Majumdar88] Shikharesh Majumdar, Derck L. Eager, and Richard B. Bunet, "Scheduling in Multiprogrammed Parallel Systems," *ACM SIGMETRICS Conference*, (May 1988) 104-113.
- [Mandyam92] S. Mandyam and Mansur H. Samadzadeh, "Scheduling Algorithms for Precedence Graphs," *Proceedings of the 1992 ACM / SIGAPP Symposium on Applied Computing (SAC' 92)*, Kansas City, MO, (March 1992) 747-756.
- [Minsky70] M. Minsky, "Form and Computer Science," *ACM Turing Lecture, JACM*, Vol. 17, No. 2, (February 1970) 197-215.
- [Nutt92] Gary J. Nutt, *Centralized and Distributed Operating Systems*, Prentice Hall, Englewood Cliffs, NJ 1992.
- [Polychronopoulos86] C. D. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. Dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, Champaign, IL, 1986.
- [Polychronopoulos88] C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishing, Norwel, MA, 1988.
- [Sahni76] S. Sahni, "Algorithms for Scheduling Independent Tasks," *JACM*, No. 23, (1976) 116-127.
- [Samadzadeh91] Farideh A. Samadzadeh, "Implementation of Cooperating and Computing Algorithms on Sequent S/81," *The 29th ACM Southeast Regional Conference*, Auburn, AL, (April 1991) 356-358.
- [Samadzadeh92a] F. Samadzadeh and G. E. Hedrick, "Near-Optimal Multiprocessor Scheduling," *The Proceedings of The 1992 ACM Computer Science Conference*, Kansas City, MO (1992) 477-484.
- [Samadzadeh92b] F. Samadzadeh and G. E. Hedrick, "A Heuristic Multiprocessor Scheduling Algorithm for Creating Near-Optimal Schedules Using Task System Graphs," *The Proceedings of The 1992 ACM Symposium on Applied Computing*, Kansas City, MO (1992) 711-718.
- [Samadzadeh92c] Farideh A. Samadzadeh, "Scheduling Algorithms for Parallel Execution of Computer Programs," Ph.D. Dissertation, Computer Science Department, Oklahoma State University, Stillwater, OK, (July 1992).

- [Samadzadeh 93] F. Samadzadeh, A. Thobani, and M. Samadzadeh, "Implementation and Performance Evaluation of a Self-Timed Multiprocessor Scheduler on a Shared-Memory Machine," Submitted to *The Supercomputing'93 Conference*, Portland, OR, (November 1993).
- [Sarkar89] Vivek Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, The MIT Press, Cambridge, MA, 1989.
- [SEQ90] *Symmetry Multiprocessor Architecture Overview*, Sequent Computer Systems, Inc., 1990.
- [SEQ92] *Guide to Parallel Programming on the Sequent Computer Systems*, Prentice Hall, Englewood Cliffs, NJ, 1992.
- [Shani76] S. Shani, "Algorithms for Scheduling of Independent Tasks," *JACM*, No. 23 (1976) 116-127.
- [Thobani92] A. Thobani and M. Samadzadeh, "Scheduling of Independent Tasks on Sequent Symmetry S/81," *Proceedings of the 5th Annual Conference: Sequent Users' Resource Forum (SURF '92)*, Atlanta, GA (October 1992) 217-236.
- [Ullman75] J. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Science*, Vol. 10 (1975) 384-393.

APPENDIX A

GLOSSARY AND TRADEMARK INFORMATION

GLOSSARY

- Adjacency Matrix:** Boolean matrix indicating, for each pair of vertices i and j , whether there is an edge from i to j .
- ATT:** Average Turnaround Time, the sum of all turnaround times divided by the number of tasks; Turnaround Time is defined as the sum of the processing time (i.e., time spent on the processor), the waiting time (i.e., time spent out of the processor in the ready queue and other queues), and the input/output (I/O) time.
- Busy-Waiting:** Using processor cycles to test a variable actively and repeatedly until it assumes a desired value.
- Condition Synchronization:** Delaying the continued execution of a process until some data object it shares with another process is in an appropriate state.
- Directed Acyclic Graph:** A graph without any cycles in which the edges have an orientation, denoted by arrowheads.
- Directed Graph:** A graph in which the edges have orientations denoted by arrowheads.
- Dynamic Scheduling:** A scheduling method that assigns tasks to processors at run-time, i.e., it is not known till run-time which processor will process which task.
- Edge:** A component of a graph. An edge is a pair of vertices. If the edge is directed, the pair is ordered; if the edge is undirected, the pair is unordered.
- Efficiency:** Ratio of speedup to number of processors used.

- ESP:** Earliest Schedule Partition; dividing a task graph into a number of partitions such that each of the tasks are placed in a partition based on the earliest time they can be scheduled.
- Job:** A system command, a user program, or a task given to a scheduler for scheduling.
- MIMD:** Multiple-instruction stream, multiple data stream.
- Multiprogramming:** Allowing more than one program to be in some state of execution (not necessarily executing) at the same time.
- Parallelism:** The use of multiple resources to increase concurrency.
- Parallel Processing:** A type of information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem.
- Prescheduled:** A type of partitioning algorithm in which each processor is allocated its share of the computation at compile-time.
- Process Flow Graph:** An directed acyclic graph in which vertices represent processes and edges represent execution constraints.
- Ready List:** An operating system list containing ready-to-run processes.
- Schedule:** An allocation of tasks to processors.
- SIMD:** Single-instruction stream, multiple data stream.
- Self-Scheduled:** A type of partitioning algorithm in which work is assigned to processors dynamically at run-time.
- Speedup:** Time taken to execute the best sequential algorithm solving a problem divided by the time taken to execute a parallel algorithm solving the same problem.
- Static Scheduling:** A scheduling method that assigns tasks to processors at run time in a predetermined fashion.

Task Partitioning: The division of the tasks in a task system among the available processors in a multiprocessor machine.

Timesharing: A type of multiprogramming that allows a number of users to interact with their programs in real time.

Upper Triangular: A matrix with no nonzero element below the main diagonal.

TRADEMARK INFORMATION

DYNIX/ptx: A registered trademark of Sequent Computer Systems, Inc.

DYNIX/ptx is an operating system for Sequent Computers.

Ethernet: A registered trademark of Xerox Corporation.

Intel: A registered trademark of Intel Corporation.

Symmetry S/81: A registered trademark of Sequent Computer Systems.

UNIX: A registered trademark of AT&T.

APPENDIX B

GRAPHS

TERMINATION SEQUENCE

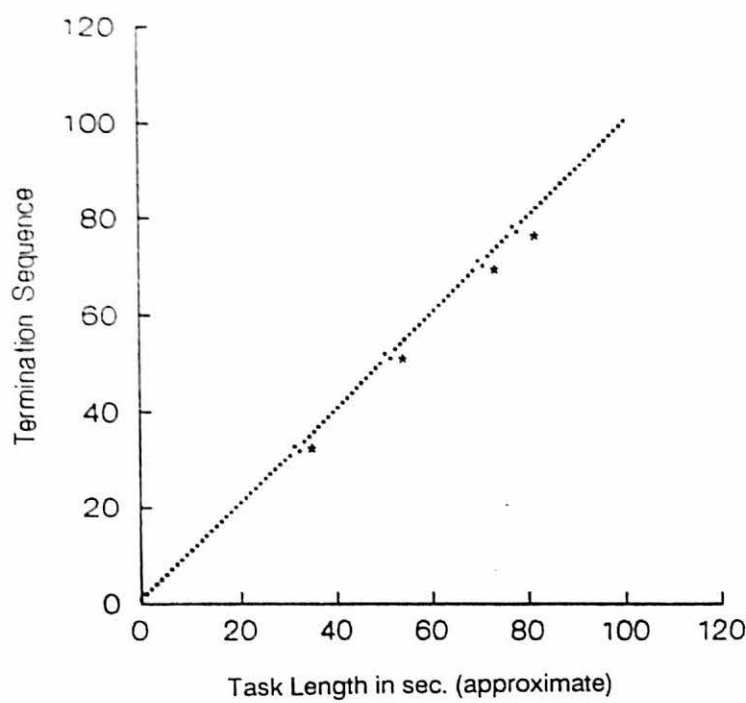


Figure 1.a

GANTT CHART

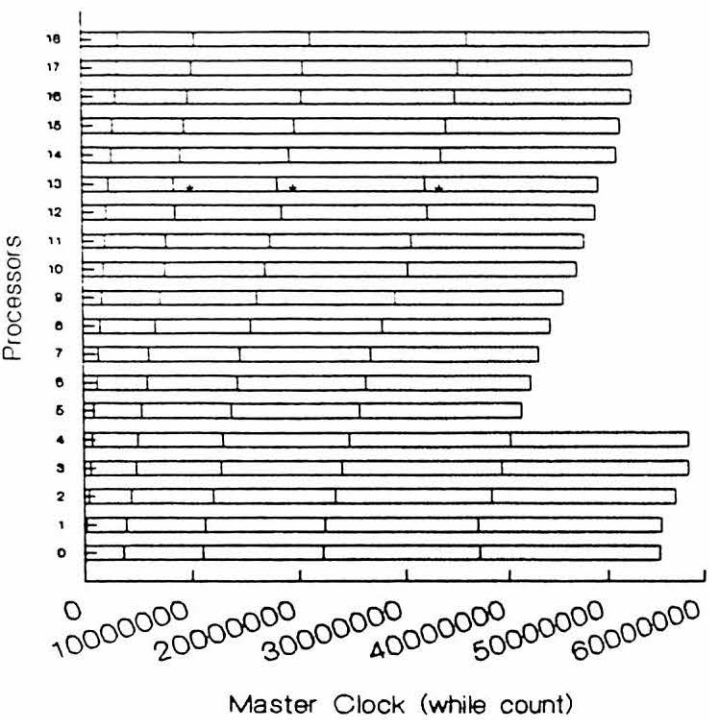


Figure 1.b

PROGRESS CHART

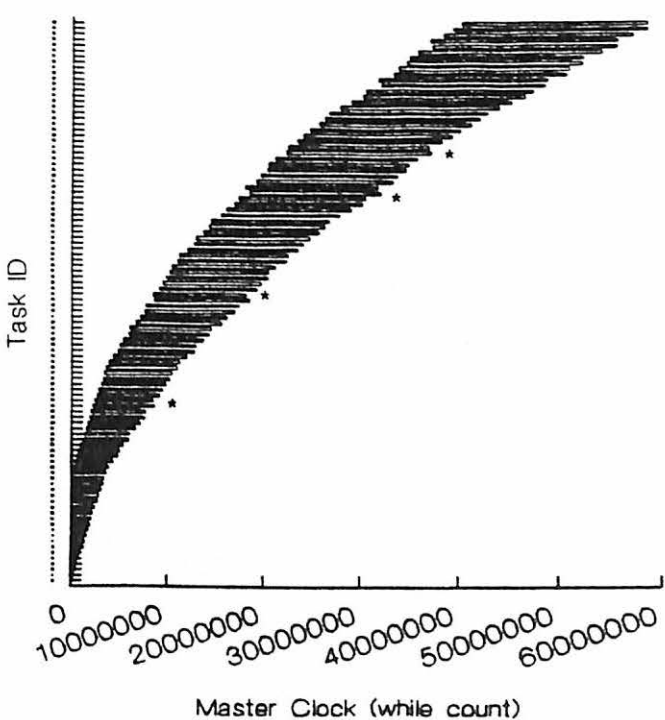


Figure 1.c

Task Set One (19, 100, WHILE, INCR)

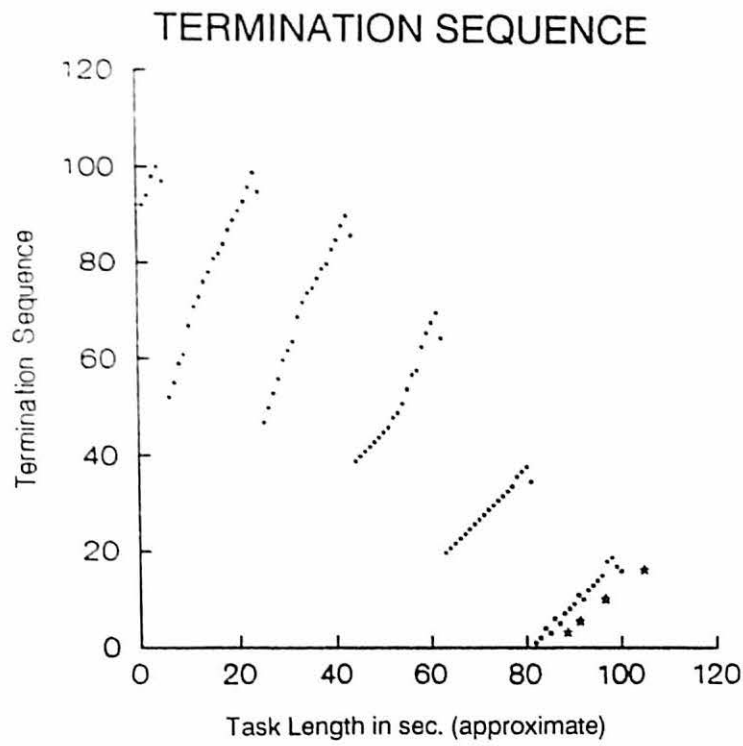


Figure 2.a

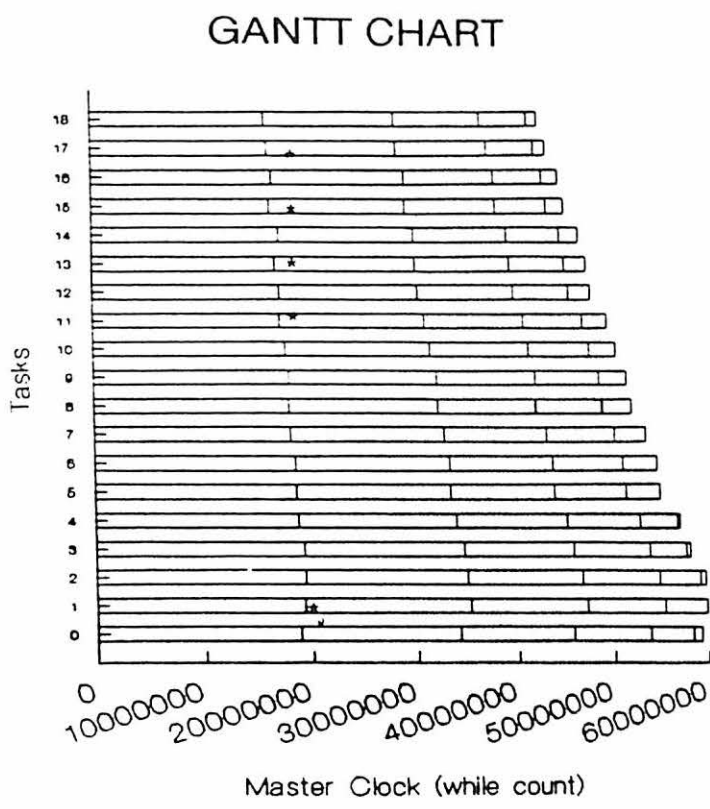


Figure 2.b

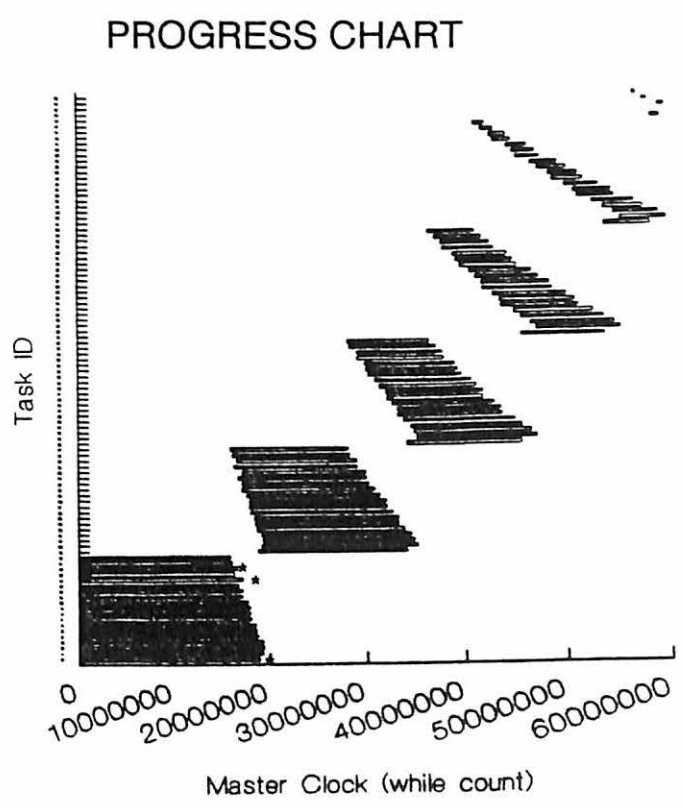


Figure 2.c

Task Set Two (19, 100, WHILE, DECR)

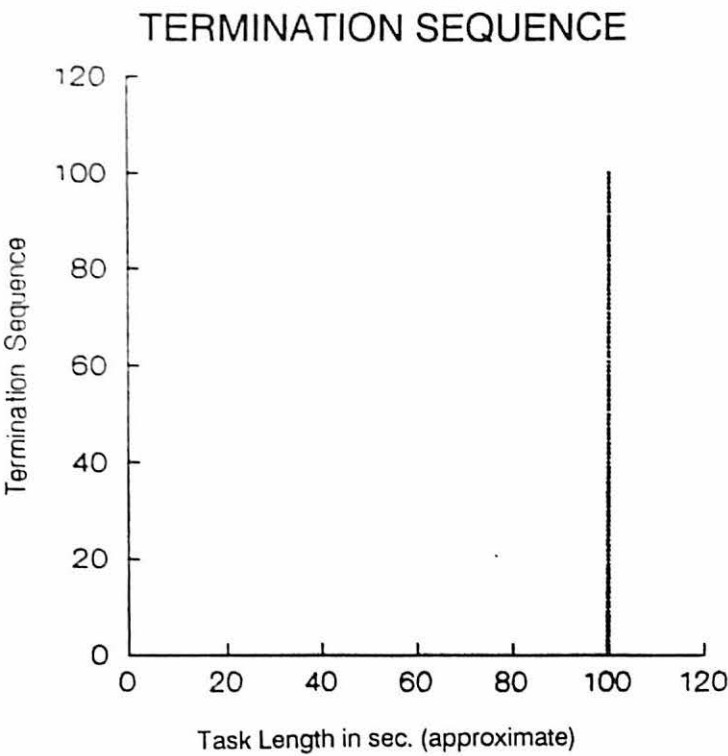


Figure 3.a

GANTT CHART

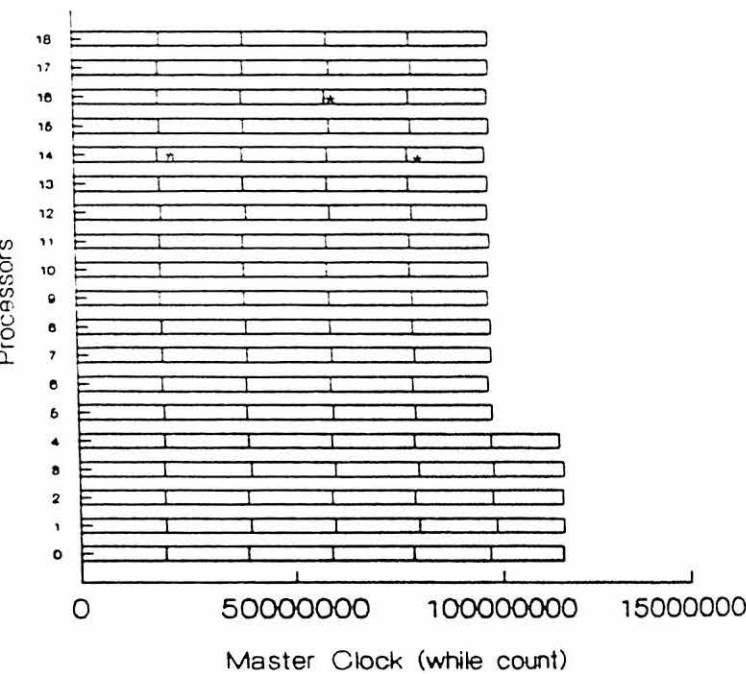


Figure 3.b

PROGRESS CHART

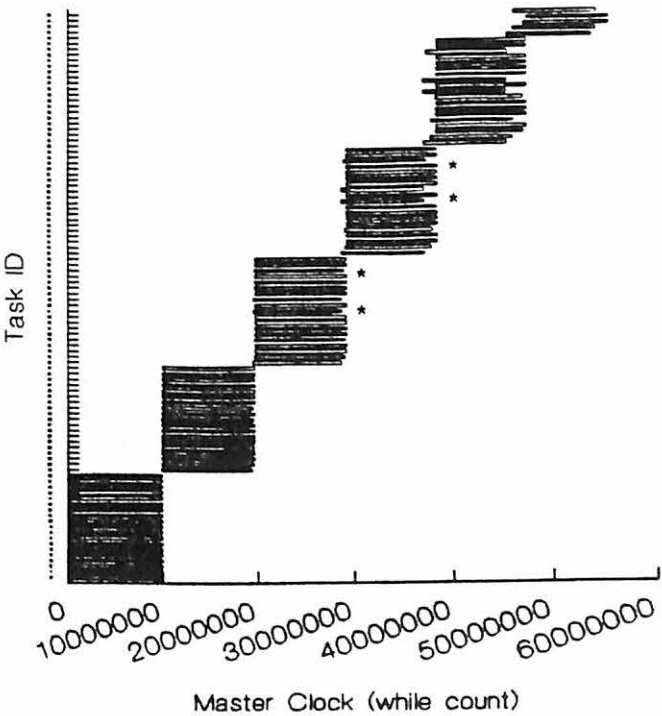


Figure 3.c

Task Set Three (19, 100, WHILE, EQUAL)

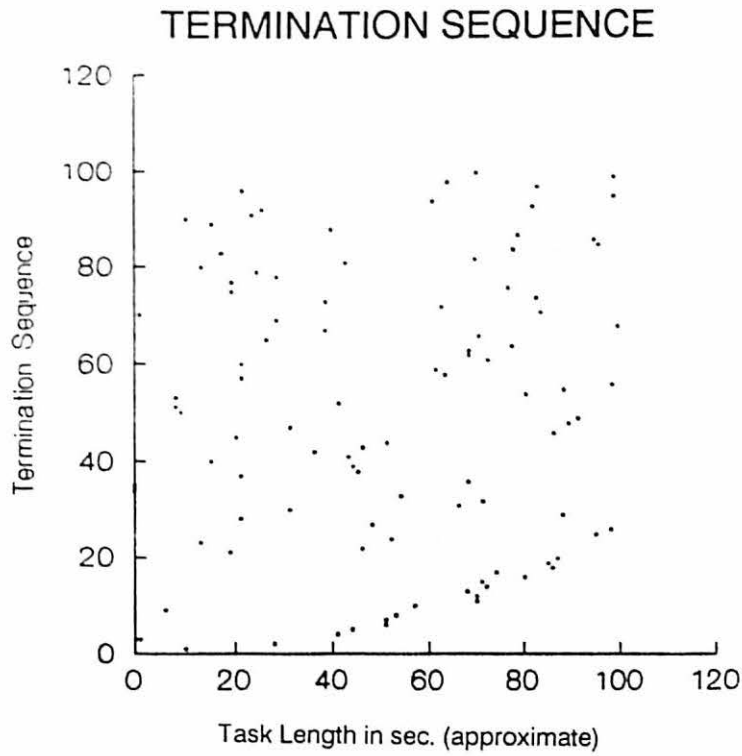


Figure 4.a

GANTT CHART

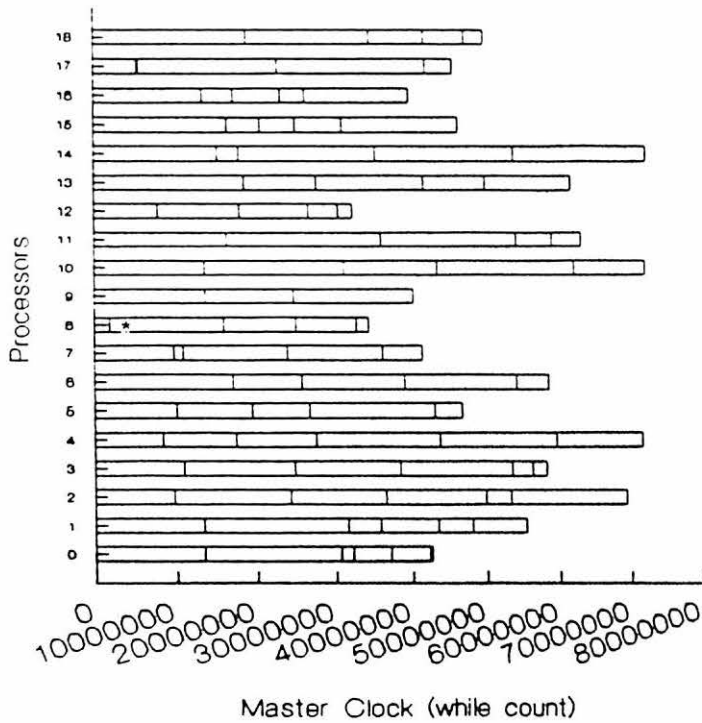


Figure 4.b

PROGRESS CHART

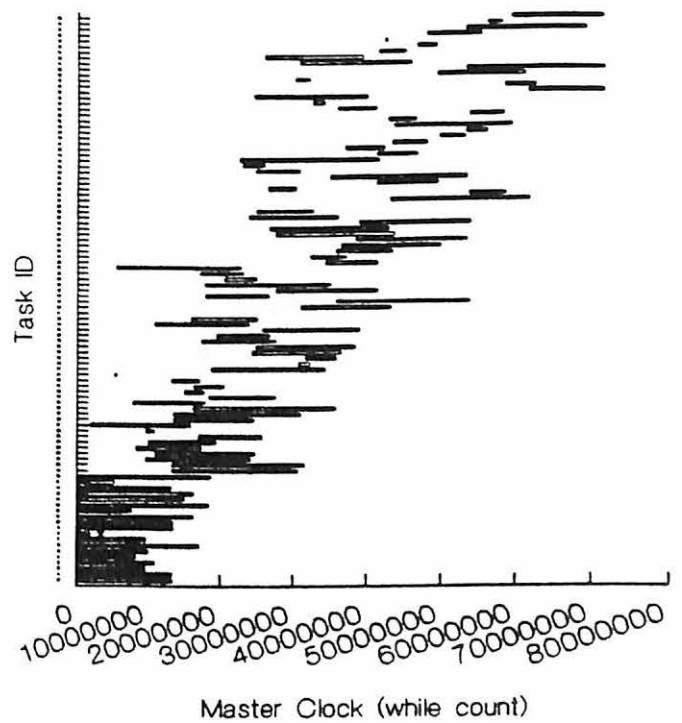


Figure 4.c

Task Set Four (19, 100, WHILE, RAND)

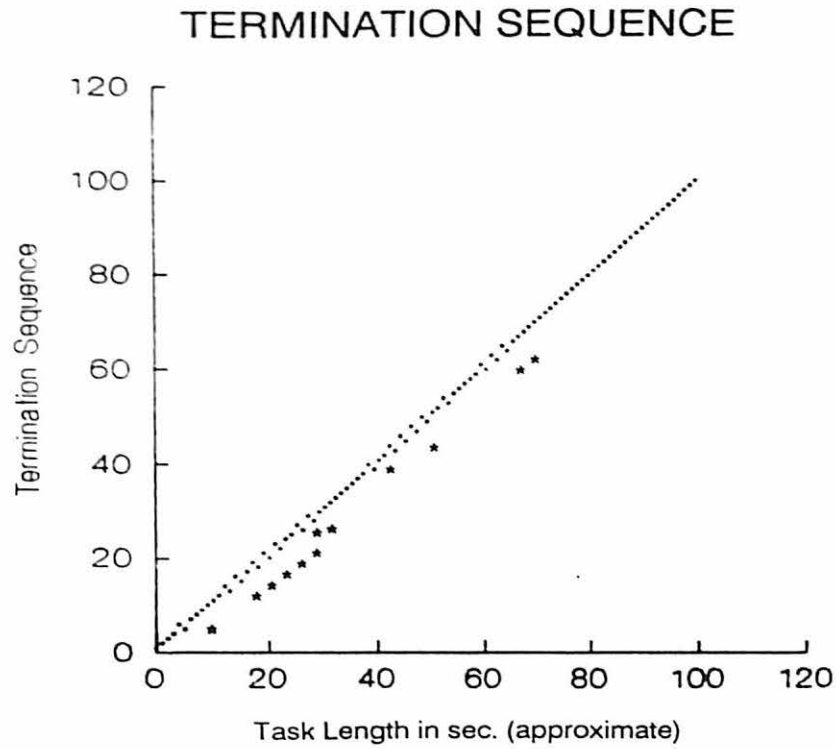


Figure 5.a

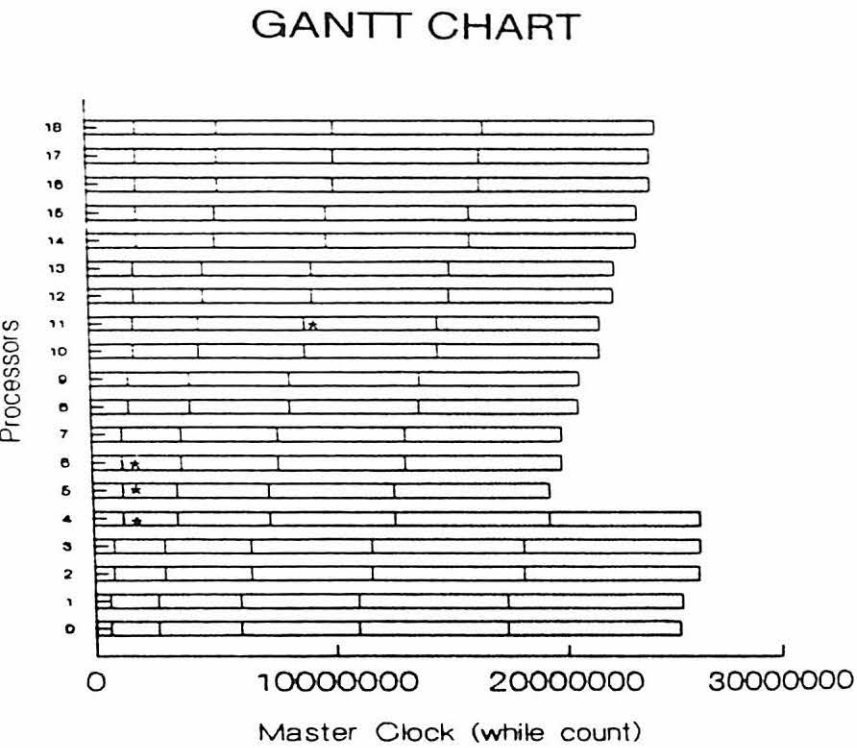


Figure 5.b

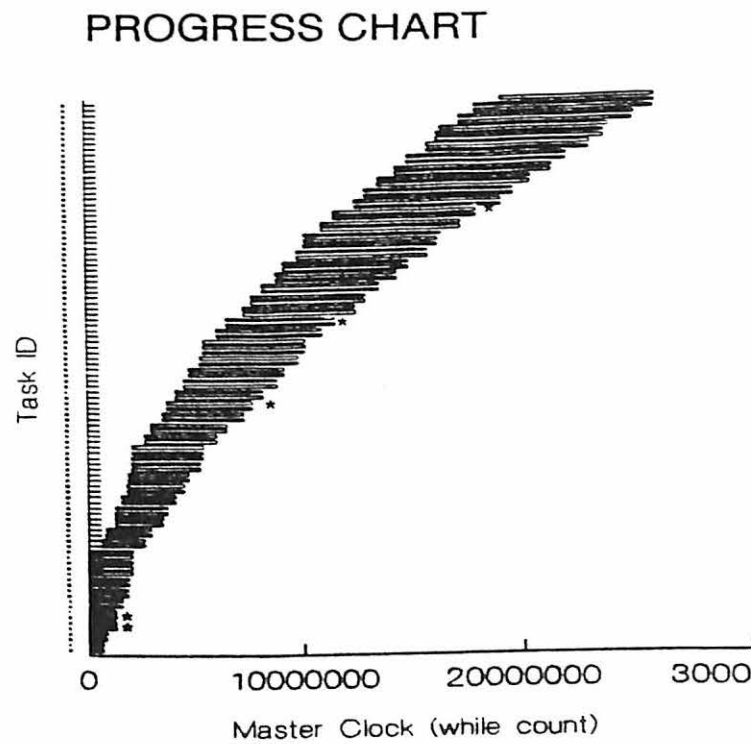


Figure 5.c

Task Set Five (19, 100, SLEEP, INCR)

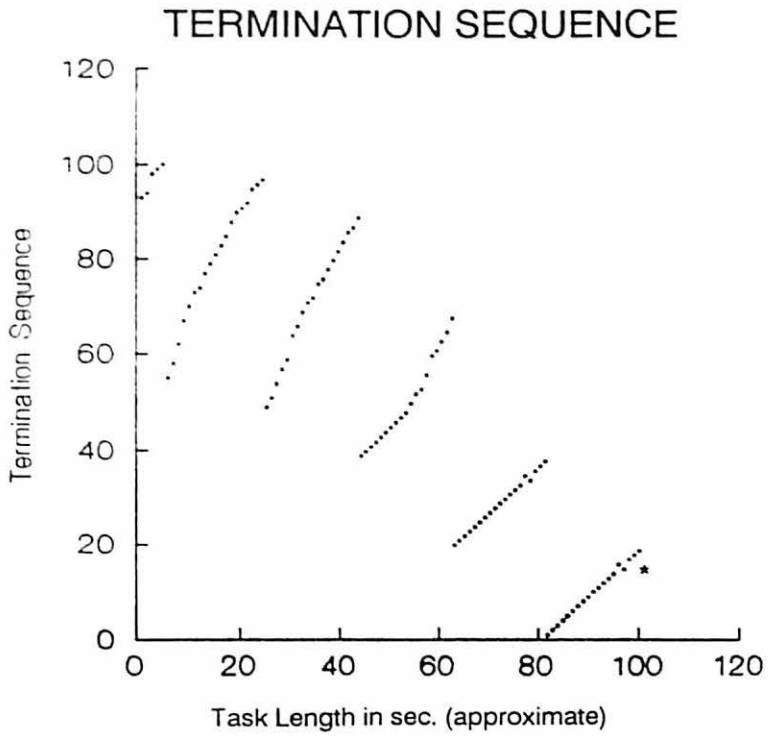


Figure 6.a

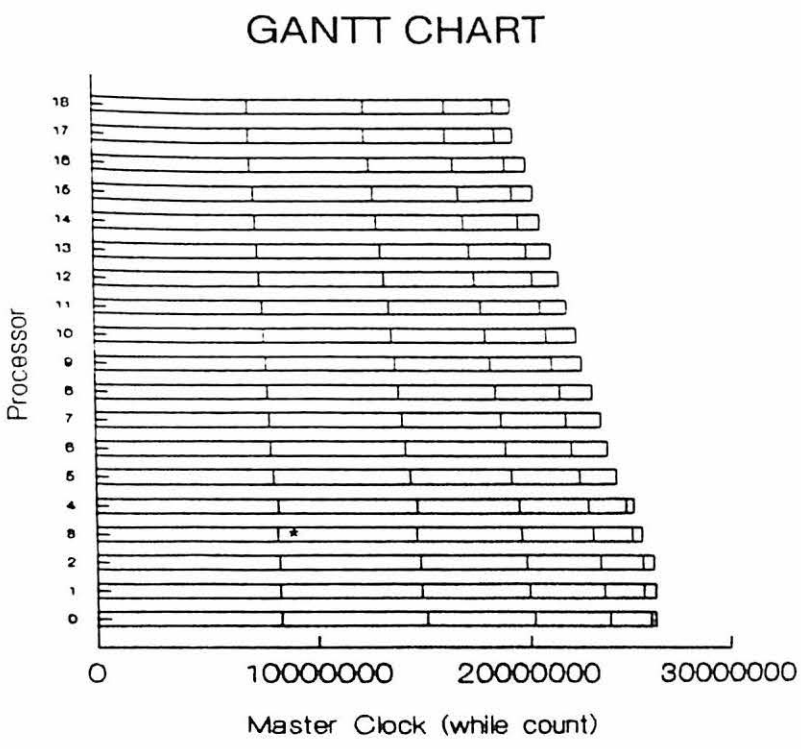


Figure 6.b

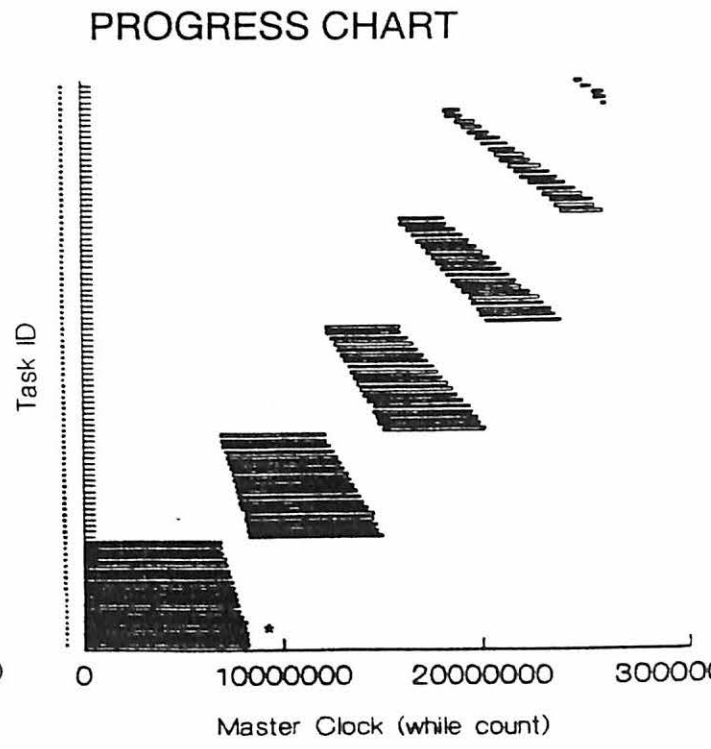


Figure 6.c

Task Set Six (19, 100, SLEEP, DECR)

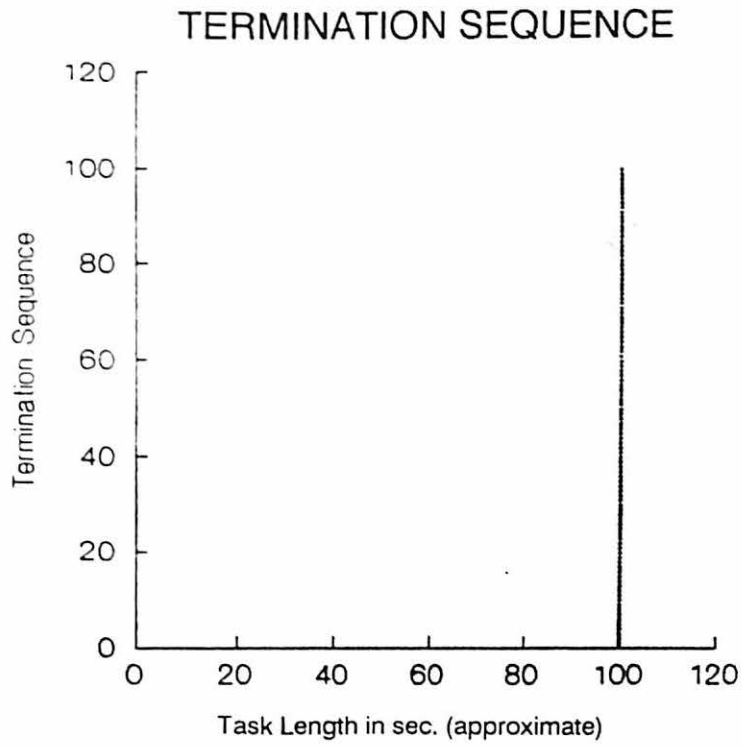


Figure 7.a

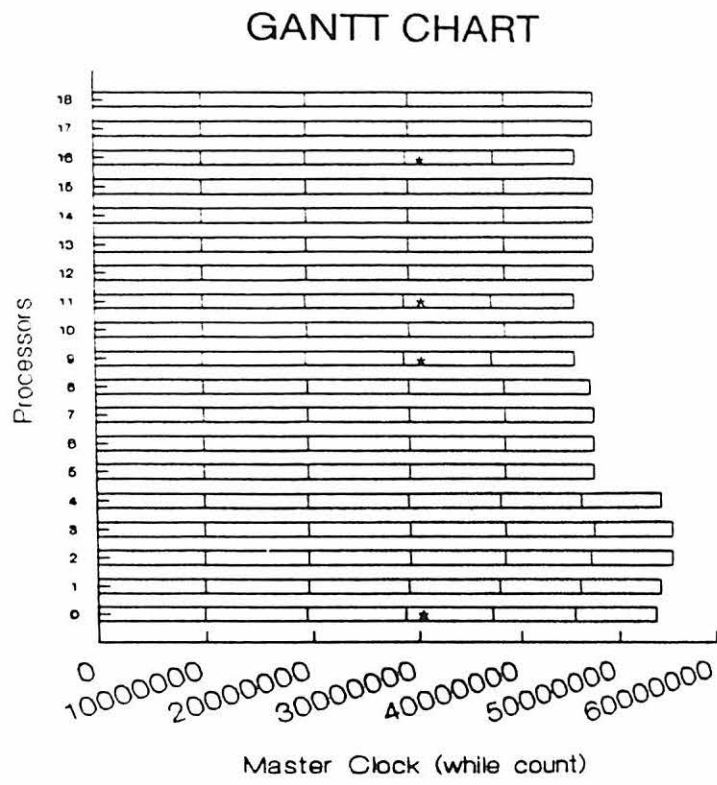


Figure 7.b

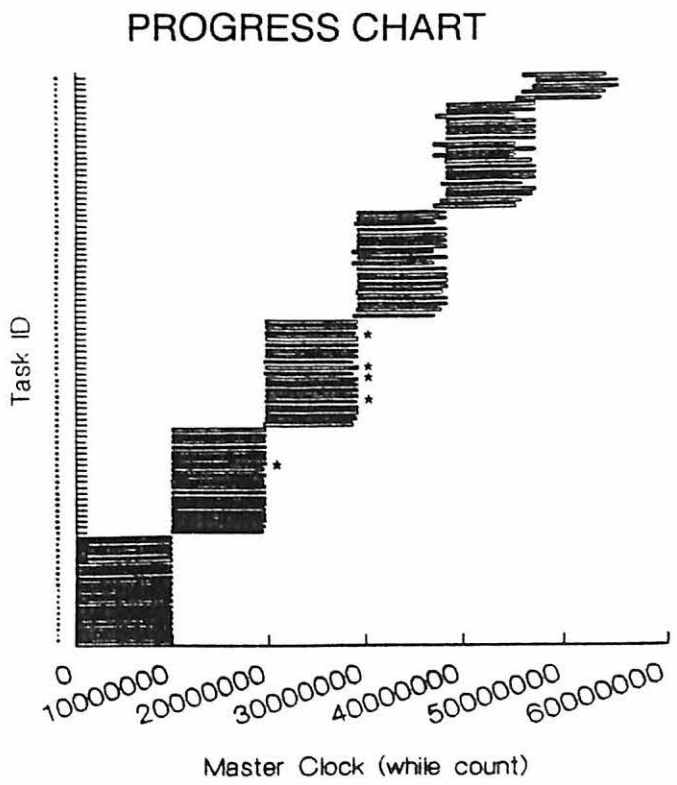


Figure 7.c

Task Set Seven (19, 100, SLEEP, EQUAL)

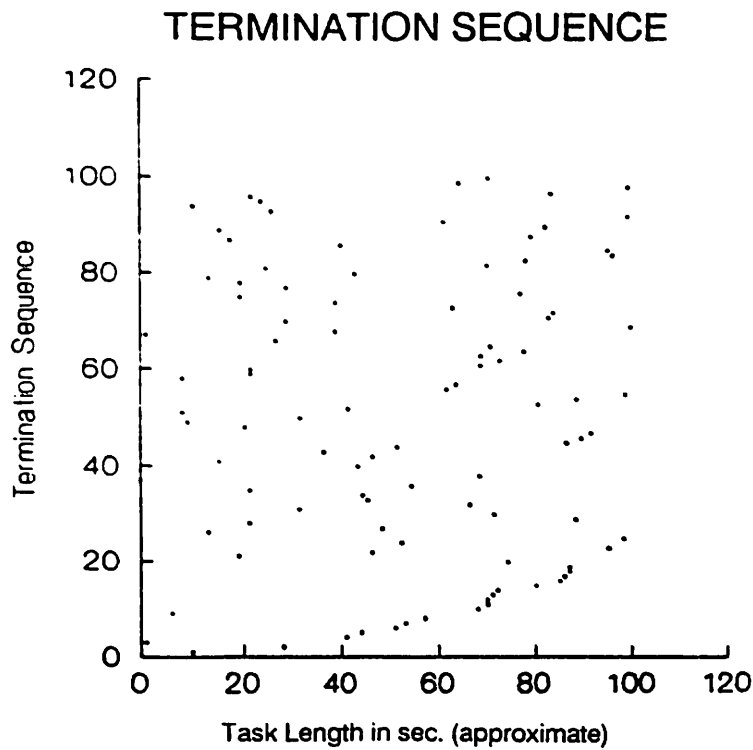


Figure 8.a

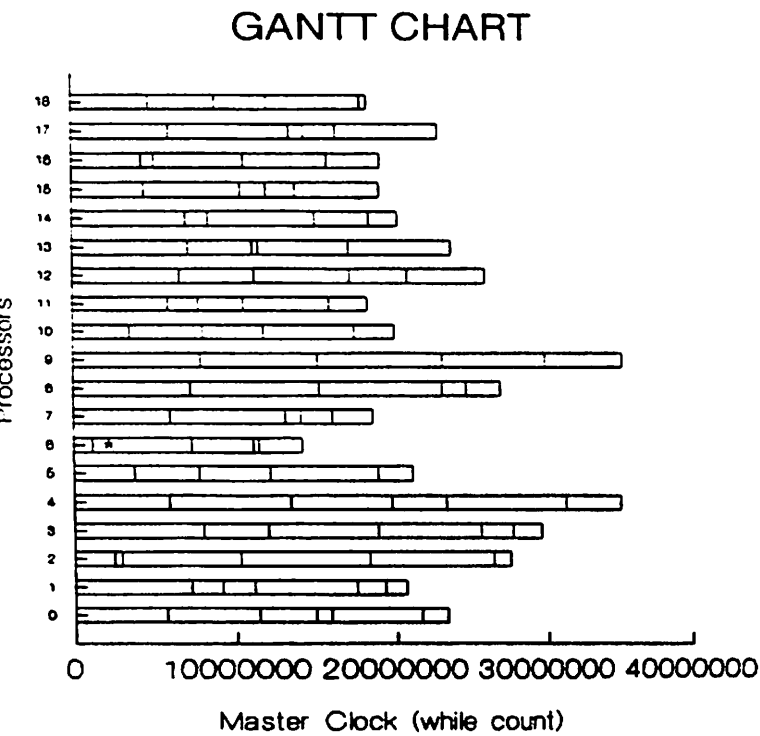


Figure 8.b

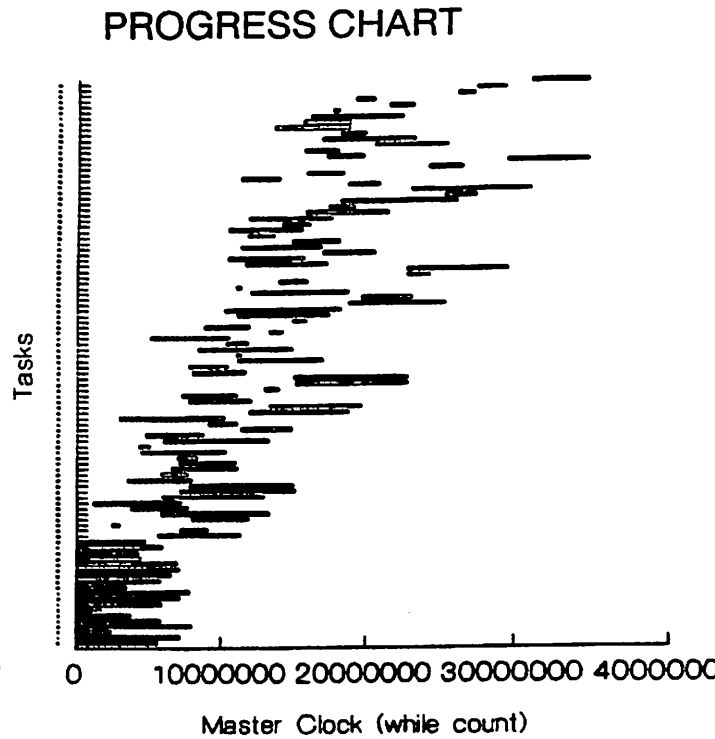


Figure 8.c

Task Set Eight (19, 100, SLEEP, RAND)

APPENDIX C

PROGRAM LISTING

The Self Timed Scheduler consists of 8 program files. The order of the program listings is as follows:

1. indtg.c
2. deptg.c
3. esp.c
4. vl.c
5. daf.c
6. rw.c
7. vl_esp.c
8. ss_dep.c

```

/*=====
 *
 *                               indtg.c
 *=====*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "set.h"

#define TRUE      1
#define FALSE     0
#define a         16807.0      /* used in the random number generator */
#define m         2147483647.0 /* used in the random number generator */
#define q         127773.0     /* used in the random number generator */
#define r         2836.0       /* used in the random number generator */

double random();
int    normalize();

/*
 *-----
 *                               main()
 *-----
 * In order to run indtg.c following command should be given at system prompt
 *
 * $ indtg N LMIN LMAX > task_file
 * where
 *
 *      N: number of tasks
 *      LMIN: minimum task length
 *      LMAX: maximum task length
 *      task_file: name of the task system file (output)
 *-----
 */
main(argc, argv)
int  argc;
char *argv[];
{
    double  seed,          /* used in the random number generator */
           primer;
    int  i,                /* loop counter */
        upper,            /* upper bound on the task length */
        lower,            /* lower bound on the task length */
        n_tasks;          /* number of tasks in the task system */

    /*
     * check for the correct number of parameters
     */
    if (argc != 4) {
        printf("*** Error: Invalid number of parameters \n");
        exit(1);
    }

    /*
     * initialize variables for the random number generator
     */
    seed = 2.009;
    for (i=1; i<=100; i++)
        primer = random(&seed);

```

```

/*
 * initialize variables
 */
n_tasks = atoi(argv[1]);
lower = atoi(argv[2]);
upper = atoi(argv[3]);

/*
 * generate task system
 */
for (i=1; i<=n_tasks; i++)
    printf("%d\n", normalize(&seed, lower, upper));
}

/*
-----
 * rand_gene()
-----
 * generate a random number between 0 and 1
-----
 */
double random(seed)
double *seed;          /* used by the random number generator */
{
    double rand,        /* random number generated */
           lo,          /* used by random number generator */
           hi,          /* used by random number generator */
           test;        /* used by random number generator */
    int    tmp_int;      /* used by random number generator */

    /*
     * generate a random number
     */
    tmp_int = *seed/q;
    hi      = tmp_int*(1.0);
    lo      = *seed - q*hi;
    test    = a*lo - r*hi;
    if (test > 0.0)
        *seed = test;
    else
        *seed = test + m;
    rand = *seed/m;
    return(rand);
}

/*
-----
 * normalize()
-----
 * find a task length between "lower" and "upper"
-----
 */
int normalize(seed, lower, upper) /* seed used by the random number generator */
double *seed;                   /* minimum value of length */
int    lower,                   /* maximum value of length */
       upper;
{
    long  N,
           temp,
           norm;
    double temp2;

    /*
     * find a length
     */
    temp2 = -log(random(seed))/2.38907;
    while (temp2 > 1)
        temp2 = temp2 - 1.0;

    temp = m*temp2;
    N = temp % upper;

    if (N == 0)
        norm = upper;
    else if (N < lower)
        norm = lower;
    else
        norm = N;
    return norm;
}

```

```

/*=====
 *
 *                               deptg.c
 *=====*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "set.h"

#define TRUE      1
#define FALSE    0
#define a        16807.0      /* used by the random number generator */
#define m        2147483647.0 /* used by the random number generator */
#define q        127773.0     /* used by the random number generator */
#define r        2836.0       /* used by the random number generator */

int  adj_mtx[MAX_TASKS][MAX_TASKS]; /* adjacency matrix */
    task_set[MAX_TASKS];           /* task system */

double random();
void  init_adj_mtx();
void  create_source();
void  print_adj_mtx();

/*
 *-----
 *                               main()
 *-----
 * In order to run deptg.c following command should be given at system prompt.
 * $  deptg N P > adj_file
 *   where
 *       N: number of tasks
 *       P: degree of parallelism
 *   adj_file: adjacency matrix created (output)
 *-----
 */
main(argc, argv)
int  argc;
char *argv[];
{
    double seed, primer, /* used by the random number generator */
           p1, p2;
    int    n_tasks,      /* number of tasks in the task system */
           ok,
           i, j,          /* loop counters */
           task;          /* a task */

    /*
     * check for the correct number of parameters
     */
    if (argc != 3) {
        printf("*** Error: Invalid number of parameters \n");
        exit(1);
    }

    /*
     * get the number of tasks and degree of parallelism
     */
    n_tasks = atoi(argv[1]);
    p1 = atof(argv[2]);

    /*
     * initialize the random number generator
     */
    seed = 2.009;
    for (i=1; i<=100; i++)
        primer = random(&seed);

    /*
     * initialize adjacency matrix
     */
    init_adj_mtx(adj_mtx, task_set, n_tasks);

    /*
     * create adjacency matrix
     */
    for (task=2; task<= n_tasks-1; task++)
        for (j=task; j<=n_tasks; j++) {
            p2 = random(&seed);

```



```

        if (p2 <= p1)
            adj_mtx[task][j] = 1;
        else
            adj_mtx[task][j] = 0;
    }

    for (j=1; j<=n_tasks; j++)
        adj_mtx[j][j] = 0;

    adj_mtx[n_tasks-1][n_tasks] = 1; /* define sink, sink is the last node */
                                    /* to be executed. */
    for (i=2; i<= n_tasks-1; i++) {
        ok = 0;
        for (j=i; j<=n_tasks; j++)
            if (adj_mtx[i][j] == 1)
                ok = 1;
        if (!ok)
            adj_mtx[i][n_tasks] = 1;
    }

    /*
     * create source node
     */
    create_source(adj_mtx, n_tasks);

    /*
     * print the adjacency matrix
     */
    print_adj_mtx(adj_mtx, n_tasks);
}

/*
 *-----
 *                               print_adj_mtx()
 *-----
 * print the adjacency matrix created
 *-----
 */
void print_adj_mtx(adj_mtx, n_tasks)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    n_tasks; /* number of tasks */
{
    int i, j, k; /* loop counters */

    /*
     * print adjacency matrix
     */
    for (i=1; i<=n_tasks; i++) {
        for(k=1; k<=i; k++)
            printf(" ");
        for (j=i; j<=n_tasks; j++)
            printf("%d", adj_mtx[i][j]);
        printf("\n");
    }
}

/*
 *-----
 *                               create_source()
 *-----
 * create the source node
 *-----
 */
void create_source(adj_mtx, n_tasks)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    n_tasks; /* number of tasks */
{
    int row, col, i, j, all_zero;

    /*
     * initialize the first row of adjacency matrix to 0
     */
    for (i=1; i<= n_tasks; i++)
        adj_mtx[1][i] = 0;

    /*
     * create the source node
     */
    row = 2;
    col = 2;

```

```

for (i=2; i<= (n_tasks-1); i++) {
    all_zero = TRUE;
    for (j=2; j<=row; j++)
        if (adj_mtx[j][col] == 1)
            all_zero = FALSE;
    if (all_zero)
        adj_mtx[1][col] = 1;
    row++;
    col++;
}
}

/*
-----
*
*                               init_adj_mtx()
*-----
* initialize the adjacency matrix
*-----
*/
void init_adj_mtx(adj_mtx, task_set, max_tasks)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    task_set[MAX_TASKS],           /* task system */
    max_tasks;                     /* maximum number of tasks */
{
    int i, j;

    for (i=1; i<= max_tasks; i++) {
        for (j=1; j<=max_tasks; j++)
            adj_mtx[i][j] = 0;
        task_set[i] = 0;
    }
}

/*
-----
*
*                               rand_gene()
*-----
* generates a random number between 0 and 1
*-----
*/
double random(seed)
double *seed; /* used by random number generator */
{
    double rand, /* random number generated */
           lo,   /* used by the random number generator */
           hi,   /* used by the random number generator */
           test; /* used by the random number generator */
    int tmp_int; /* used by the random number generator */

    tmp_int = *seed/q;
    hi = tmp_int*(1.0);
    lo = *seed - q*hi;
    test = a*lo - r*hi;
    if (test > 0.0)
        *seed = test;
    else
        *seed = test + m;
    rand = *seed/m;
    return(rand);
}

/*=====
*
*                               esp.c
*=====*/
#include "set.h"

#define FALSE    0
#define TRUE     1

void read_tasks();
int set_elem();
void set_copy();

int adj_mtx[MAX_TASKS][MAX_TASKS];
/*
-----
*
*                               main()
*-----

```

```

* In order to run esp.c following command should be given at system prompt.
*
* $ esp adj_mtx > layers
*   where
*       adj_mtx: name of the file having adjacency matrix
*       layers: name of the file that will have ESP layers
*-----
*/
main(argc, argv)
int argc;
char *argv[];
{
    int n_tasks,          /* number of tasks */
        i, k, v, u,      /* loop counters */
        pred,            /* if TRUE, the task has a predecessor */
        index1, index2; /* counters */
    set V, V1,            /* tasks in the task system and their copies */
        layer[MAX_TASKS]; /* ESP layers */

    /*
     * read adjacency matrix
     */
    read_tasks(argv[1], adj_mtx, &n_tasks);

    /*
     * initialize the set, and then add tasks in the set having task system
     */
    set_init(&V);
    for (i=1; i<=n_tasks; i++)
        set_add(i, &V);

    /*
     * make a copy of the task system set
     */
    k = 0;
    set_copy(V, &V1);

    /*
     * form ESP layers
     */
    while (V1.size > 0) {
        k++;
        set_init(&layer[k]);
        index1 = 1;
        v = set_elem(V1, index1);
        while (v != 0) {
            /*
             * find a task which has no predecessor
             */
            if (!predecessor(adj_mtx, v)) {
                set_add(v, &layer[k]);
                set_del(v, &V1);
                index1 = 0;
            }
            index1++;
            /*
             * find next task
             */
            v = set_elem(V1, index1);
        }
        /*
         * remove the current task as a predecessor of any task in the
         * task system
         */
        i = 1;
        while (i<=layer[k].size) {
            v = set_elem(layer[k], i);
            index2 = 1;
            u = set_elem(V1, index2);
            while (u != 0) {
                if (is_pred(adj_mtx, v, u))
                    remove_pred(adj_mtx, v, u);
                index2++;
                u = set_elem(V1, index2);
            }
            i++;
        }
    }

    /*
     * print the ESP layers created

```

```

    */
    for (i=1; i<=k; i++) {
        set_print(layer[i]);
    }
}

/*
-----
*
* remove_pred()
-----
* remove the current task from the predecessor list of all the tasks
* in the task system
-----
*/
remove_pred(adj_mtx, v, u)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    v, u;
{
    adj_mtx[v][u] = 0;
}

/*
-----
*
* is_pred()
-----
* if v is the predecessor of u
-----
*/
int is_pred(adj_mtx, v, u)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    v, u;
{
    if (adj_mtx[v][u] == 1)
        return TRUE;
    else
        return FALSE;
}

/*
-----
*
* predecessor()
-----
* find if v has any predecessor which is not done
-----
*/
int predecessor(adj_mtx, v)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    v; /* task id */
{
    int i, /* loop counter */
        pred; /* if TRUE, predecessors are not done */

    i = 1;
    pred = FALSE;
    while ((i<v) && (!pred)) {
        if (adj_mtx[i][v] == 1)
            pred = TRUE;
        i++;
    }
    return pred;
}

/*
-----
*
* read_tasks()
-----
* read adjacency matrix
-----
*/
void read_tasks(task_file, adj_mtx, n_tasks)
char *task_file; /* name of the adjacency matrix file */
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    *n_tasks; /* number of tasks */
{
    FILE *tftp; /* file pointer to the adjacency matrix file */
    char instr[MAX_TASKS+1], sch[3];
    int task_cnt, /* task counter */
        succ_cnt; /* successor count */
}

```

```

/*
 * open adjacency matrix file
 */
if ((tfp = fopen(task_file, "r")) == NULL) {
    printf("*** Error: Task file name not given\n");
    exit(1);
}

task_cnt = 0;
/*
 * read one row of the adjacency matrix
 */
fgets(instr, MAX_TASKS+1, tfp);
while (!feof(tfp)) {
    task_cnt++;
    succ_cnt = 1;
    /*
     * convert row to required format
     */
    while (instr[succ_cnt] == ' ') succ_cnt++;
    while (instr[succ_cnt] != '\0') {
        sch[0] = instr[succ_cnt-1];
        sch[1] = '\0';
        adj_mtx[task_cnt][succ_cnt] = atoi(sch);
        succ_cnt++;
    }
    /*
     * read next row
     */
    fgets(instr, MAX_TASKS+1, tfp);
}
n_tasks = task_cnt;
}

/*=====
 *                               vl.c                               *
 *=====*/
#include "set.h"

#define FALSE    0
#define TRUE     1
#define OUT_FILE "vl.sch"      /* schedule file name */

void read_weight();
void select_sort();
int  calc_LB();
void schedule();
void print_export_data();
void collect_data();
void print_adj_mtx();

/*
 * -----
 *                               main()
 * -----
 * In order to run vl.c following command should be given at system prompt.
 *
 * $ vl task_file P out_file
 *   where
 *     task_file: name of the file having a task system. This is created by
 *                 calling the task system generator i.e. "indtg". In order
 *                 to execute "indtg" following command should be given
 *                 at the $ prompt. i.e.,
 *
 *                 $ indtg T min_len max_len > task_file
 *                 where
 *                   T: number of tasks
 *                 min_len: minimum length of a task
 *                 max_len: maximum length of a task
 *
 *     P: number of processors
 *     out_file: name of the output file
 *
 * The outputs of this program are in two files
 * 1. vl.sch : have the static schedule created by variant load algorithm
 * 2. out_file: have data collected as a result of this static schedule
 *               this out_put contains following information about the
 *               schedule.
 * 1. number of tasks in task system
 * 2. number of processors used

```

```

*-----
* Print the adjacency matrix for independent task system
*-----
*/
void print_adj_mtx(n_tasks)
int n_tasks;
{
    int i, j;          /* loop counters */
    FILE *fp;          /* file pointer to adjacency matrix file */

    /*
     * open the adjacency matrix file
     */
    if ((fp = fopen("adj_file", "w")) == NULL) {
        printf("**** Error: cannot open adj_file\n");
        exit(1);
    }

    /*
     * create an adjacency matrix for independent task system by placing
     * '0's in all rows.
     */
    for(i=1; i<=n_tasks; i++) {
        for (j=1; j<=i; j++)
            fprintf(fp, " ");
        for(j=i; j<=n_tasks; j++)
            fprintf(fp, "%d", 0);
        fprintf(fp, "\n");
    }
}

/*
*-----
*                               collect_data()
*-----
* collect following performance data
* 1. number of tasks in task system
* 2. number of processors used
* 3. total time taken by the task system
* 4. total processing time
* 5. total idle time
* 6. Lower bound of schedule
* 7. schedule length
* 8. Performance ratio
* 9. Average turnaround time
*-----
*/
void collect_data(Proc, avail_p, n_tasks, w, LB, data_file)
int Proc[MAX_PROC];          /* schedule length for each processor */
int avail_p,                 /* number of processors used */
    n_tasks,                 /* number of tasks in task system */
    w[MAX_TASKS],            /* weight of tasks in task system */
    LB;                       /* Lower bound of the schedule */
char *data_file;              /* output file name */
{
    int i, j,                 /* loop counters */
        sch_length,           /* schedule length */
        idle_tm,              /* idle time */
        proc_tm,              /* processing time */
        task_id,              /* task id */
        total_time;           /* total time */

    float av_tt,               /* average turnaround time */
           pr;                 /* performance ratio */
    FILE *ofp;                 /* output file pointer */

    /*
     * open an output file
     */
    if ((ofp = fopen(data_file, "a+")) == NULL) {
        printf("**** Error: cannot open schedule file\n");
        exit(1);
    }

    /*
     * find schedule length
     */
    sch_length = Proc[1];
    for (i=1; i<=avail_p; i++)
        if (Proc[i] > sch_length)

```

```

    sch_length = Proc[i];

/*
 * find processing time
 */
total_time = sch_length * avail_p;
proc_tm = 0;
for (i=1; i<=n_tasks; i++)
    proc_tm = proc_tm + w[i];

idle_tm = total_time - proc_tm; /* find idle time */
av_tt = (float)total_time / (float)n_tasks; /* find average turnaround tm */
pr = (float) LB/sch_length; /* find performance ratio */

/*
 * print performance information in an output file
 */
fprintf(ofp, "%7d%7d%7d%7d%7d%7d%7d%7.2f%7.2f\n", n_tasks, avail_p,
        total_time, proc_tm, idle_tm, LB, sch_length, pr, av_tt);
}

/*
 * -----
 * find_lowest_index()
 * -----
 * find the processor id of a processor whose schedule length is minimum.
 * -----
 */
int find_lowest_index(Proc, avail_p, LB)
int Proc[MAX_PROCC], /* schedule length of a processor */
    avail_p, /* number of processors used */
    LB; /* Lower bound on schedule */
{
    int i, /* loop counter */
        smallest; /* processor id of minimum sch. length proc.*/

/*
 * find processor id of a processor whose schedule length is minimum.
 */
    smallest = 1;
    for (i=2; i<=avail_p; i++)
        if (Proc[i] < Proc[smallest])
            smallest = i;
    return smallest;
}

/*
 * -----
 * schedule()
 * -----
 * Find the static schedule using Variant Load algorithm.
 * -----
 */
void schedule(w, Proc, avail_p, LB, n_tasks, P)
int w[MAX_TASKS], /* weights of tasks in task system */
    Proc[MAX_PROCC], /* schedule length of processors */
    avail_p, /* number of processors used */
    LB, /* Lower bound on schedule */
    n_tasks; /* number of tasks in the task system */
set P[MAX_PROCC]; /* static schedule created */
{
    int task_id, /* task id */
        proc_id, /* processor id */
        li_proc, /* processor id of minimum sch. length proc. */
        i, j, k; /* loop counter */

/*
 * Initialize the processing time at each processor to 0
 */
    for (i=1; i<=avail_p; i++)
        Proc[i] = 0;

/*
 * Assign tasks to each processor in ascending order of task
 * length
 */
    task_id = 1;
    while ((task_id <= avail_p) && (task_id <= n_tasks)) {
        set_add(task_id, &P[task_id]);
        Proc[task_id] = w[task_id];
    }
}

```

```

    task_id++;
}

/*
 * Repeat above step until all the tasks have been assigned
 * to the processors. If the schedule length for a processor is
 * greater then the Lower bound, do not assign any more task to it.
 */
if (task_id <= n_tasks) {
    proc_id = 1;
    while (task_id <= n_tasks) {
        while ((task_id <= n_tasks) && ((Proc[proc_id] + w[task_id]) <= LB)) {
            if ((Proc[proc_id] + w[task_id]) <= LB) {
                set_add(task_id, &P[proc_id]);
                Proc[proc_id] = Proc[proc_id] + w[task_id];
                task_id++;
            }
        }
        /*
         * If still some tasks remained to be assigned, find the processor
         * with the lowest schedule length and assign tasks to it.
         */
        if (task_id <= n_tasks) {
            if (proc_id == avail_p) {
                li_proc = find_lowest_index(Proc, avail_p, LB);
                set_add(task_id, &P[li_proc]);
                Proc[li_proc] = Proc[li_proc] + w[task_id];
                task_id++;
                if (task_id < n_tasks)
                    proc_id = 1;
            }
            else
                proc_id++;
        }
    }
}

/*
-----
 *
 * print_export_data()
-----
 * Create a static schedule in the format required by the scheduler to schedule
 * on real processors.
-----
 */
void print_export_data(P, avail_p, n_tasks, w)
set P[MAX_PROC]; /* tasks assigned to a processor */
int avail_p, /* number of processors */
    n_tasks, /* number of tasks in task system */
    w[MAX_TASKS]; /* weights of tasks */
{
    int task_id, /* task id */
        proc_id, /* processor id */
        i, /* loop counter */
        index;
    char no_task[3]; /* used in getting required output format */

    strcpy(no_task, "-1");
    /*
     * generate the static schedule in the format used by the scheduler
     */
    for (task_id=1; task_id<=n_tasks; task_id++) {
        for (proc_id = 1; proc_id <= avail_p; proc_id++) {
            index = set_find(task_id, &P[proc_id]);
            if (index != 0) {
                printf("%4d", w[task_id]);
                set_del(task_id, &P[proc_id]);
            }
            else
                printf("%4s", no_task);
        }
        printf("\n");
    }
}

/*
-----
 *
 * calc_LB()
-----
 * Calculate the Lower bound for schedule

```



```

*/
int calc_LB(w, avail_p, n_tasks)
int w[MAX_TASKS],          /* weights of tasks */
    avail_p,               /* number of processors used */
    n_tasks;               /* number of tasks in task system */
{
    int sum_o_weights,      /* sum of weights */
        i,                 /* loop counter */
        LB;                /* Lower bound found */

    /*
     * initialize the sum of weight variable
     */
    sum_o_weights = 0;

    /*
     * find sum of lengths of all the tasks in the task system
     */
    for (i=1; i<=n_tasks; i++)
        sum_o_weights = sum_o_weights + w[i];

    /*
     * find the lower bound
     */
    if ((sum_o_weights / avail_p) > w[1])
        LB = sum_o_weights / avail_p;
    else
        LB = w[1];
    return LB;
}

/*
 *-----
 *                               index_of_smallest()
 *-----
 * Find a task whose length is the smallest. This procedure is used in the
 * select sort.
 *-----
 */
int index_of_smallest(w, n)
int w[MAX_TASKS],          /* weights of tasks */
    n;                     /* range to be used */
{
    int smallest, i;

    smallest = 1;
    /*
     * find the smallest length task
     */
    for (i=2; i<=n; i++)
        if (w[i] <= w[smallest])
            smallest = i;
    return smallest;
}

/*
 *-----
 *                               swap()
 *-----
 * swaps two elements, used in select sort
 *-----
 */
int swap(w, smallest, i)
int w[MAX_TASKS],          /* weights of tasks */
    smallest,              /* task id of the smallest length task */
    i;                     /* index of the task to be swapped with */
{
    int temp;

    /*
     * swap tasks at index i and index smallest.
     */
    temp = w[i];
    w[i] = w[smallest];
    w[smallest] = temp;
}

/*
 *-----
 *                               select sort()
 *-----

```

```

-----
* sorts tasks in the structure w in descending order of length
*-----
*/
void select_sort(w, n_tasks)
int w[MAX_TASKS], /* weights of tasks */
    n_tasks;      /* number of tasks in task system */
{
    int i, /* loop counter */
        smallest; /* index of the smallest lenght task */

    /*
     * does select sort by sorting tasks in descending order of their
     * length
     */
    for (i=n_tasks; i>=1; i--) {
        smallest = index_of_smallest(w, i);
        swap(w, smallest, i);
    }
}

/*
-----
* read_weight()
-----
* reads weights of tasks in the variable "w" from "task_file"
-----
*/
void read_weight(w, n_tasks, argv)
int w[MAX_TASKS], /* weights of tasks */
    n_tasks;      /* number of tasks in task system */
char *argv[];     /* name of a task system file */
{
    FILE *tfp; /* pointer to a task system file */
    char task_w[4], /* weight of a task */
        task_file[20]; /* name of a task file */
    int task_id; /* task id */

    /*
     * open a task system file
     */
    strcpy(task_file, argv[1]);
    if ((tfp = fopen(task_file, "r")) == NULL) {
        printf("*** Error: Task file name not given\n");
        exit(1);
    }

    /*
     * reads tasks from the task system file into the structure "w"
     */
    task_id = 0;
    fgets(task_w, MAX_TASKS+1, tfp);
    while (!feof(tfp)) {
        task_id++;
        w[task_id] = atoi(task_w);
        fgets(task_w, MAX_TASKS+1, tfp);
    }
    *n_tasks = task_id;
}

/*=====
*
* daf.c
*=====*/
#include "set.h"
#include <math.h>

#define FALSE 0
#define TRUE 1
#define OUT_FILE "daf.sch" /* output file that contain static */
                          /* scheduling */

void read_weight();
void print_weight();
void select_sort();
int calc_LB();
void schedule();
void print_export_data();
void collect_data();
void print_adj_mtx();
int is_odd();
void sort_set();

```

```

int Test_0();
int Test_1();

/*
-----
*                                     main()
-----
* In order to run daf.c following command should be given at $ prompt
*
* $ daf task_file P out_file
*   where
*     task_file: name of file having task system. This is created by
*                 calling task system generator i.e. "indtg". In order
*                 to execute "indtg", the following command should be given
*                 at the system prompt, i.e.,
*
*                 $ indtg T min_len max_len > task_file
*                 where
*                   T: number of tasks
*                   min_len: minimum lenght of task
*                   max_len: maximum lenght of task
*
*                   P: number of processors
*                   out_file: name of output file
*
* The outputs of this program are in two files
*   1. daf.sch : have the static schedule creates by divide & fold algorithm
*   2. out_file: have data collected as a result of this static schedule
*                 this out_put contain follwoing inforamtion about the
*                 schedule.
*                 1. number of tasks in task system
*                 2. number of processors used
*                 3. total time taken by the task system
*                 4. total processing time
*                 5. total idle time
*                 6. Lower bound of schedule
*                 7. schedule length
*                 8. Performance ratio
*                 9. Avarage turnaround time
*-----
*/
main(argc, argv)
int argc;
char *argv[];
{
    int n_tasks,          /* number of tasks in task system */
        avail_p,          /* number of processors available */
        LB,               /* Lower bound */
        i, T1, T2, M1, M2,
        t0, t1, j, k, k1,
        w[MAX_TASKS],     /* weights of tasks */
        Proc[MAX_PROC];   /* schedule length of each processor */
    set P[MAX_TASKS];     /* tasks assigned to a processor */

    /*
     * check for correct number of parameters
     */
    if (argc != 4) {
        printf("**** Error: Invalid number of parameters \n");
        exit(1);
    }

    /*
     * store number of processors to be used
     */
    avail_p = atoi(argv[2]);

    /*
     * initialize tasks assigned to processor to 0
     */
    for (i=1; i<=avail_p; i++)
        set_init(&P[i]);

    /*
     * read weight (i.e., length) of a task in weight information structure
     */
    read_weight(w, &n_tasks, argv);

    /*
     * sort tasks in the task system in ascending order of task length
     */

```

```

select_sort(w, n_tasks);

/*
 * find lower bound of schedule
 */
LB = calc_LB(w, avail_p, n_tasks);

/*
 * schedule tasks in the task system using divide & fold scheduling
 * algorithm
 */
schedule(w, Proc, avail_p, LB, n_tasks, P);

/*
 * sort tasks assigned to a processor
 */
for (i=1; i<=avail_p; i++)
    select_sort(P[i].elem, P[i].size);

/*
 * performs two test for workload balancing
 */
for (i=1; i<= avail_p/2; i++) {
    j = avail_p - i + 1;
    t0 = Test_0(P, &T1, &T2, i, j, avail_p, w);
    t1 = Test_1(w, P, T1, T2, &M1, &M2, i, j, &k, &k1, avail_p, LB);
    if (t1 < t0) {
        if (t1 == M1) {
            set_del(k, &P[i]);
            set_add(k, &P[j]);
        }
        else {
            set_del(k1, &P[i]);
            set_add(k1, &P[j]);
        }
    }
}

/*
 * print adjacency matrix for independent task system.
 */
print_adj_mtx(n_tasks);

/*
 * print the static schedule created in the format required by the
 * scheduler
 */
print_export_data(P, avail_p, n_tasks, w);

/*
 * print the performance data collected
 */
collect_data(Proc, avail_p, n_tasks, w, LB, argv[3]);
}

/*
 *-----
 *                               Test_1()
 *-----
 * Test two of second phase of D&F algorithm. It finds the values of some
 * variables that will be later used to decide which task should be
 * exchanged between two processors.
 *-----
 */
int Test_1(w, P, T1, T2, M1, M2, i, j, k, k1, avail_p, LB)
int w[MAX_TASKS];
set P[MAX_TASKS];
int T1, T2, *M1, *M2, i, j, *k, *k1, avail_p, LB;
{
    int t1, t2, smallest;

    smallest = P[i].size;
    t1 = P[i].elem[smallest];
    if ((T1 - t1) > LB)
        t1 = P[i].elem[1];
    if ((T1-t1) > (T2+t1))
        *M1 = T1-t1;
    else
        *M1 = T2+t1;
}

```

```

if (P[i].size > 1) {
    smallest = P[i].size-1;
    t2 = P[i].elem[smallest];
    if ((T1 - t2) > LB)
        t2 = P[i].elem[1];
    if ((T1-t2) > (T2+t2))
        *M2 = T1-t2;
    else
        *M2 = T2+t2;
}
else
    *M2 = 0;

*k = t1;
*k1 = t2;

if (*M1 < *M2)
    return *M1;
else
    return *M2;
}

/*
-----
*                                     Test_0()
-----
* Test one of second phase of D&F algorithm. It finds values of some
* variables that will be later used to decide which task should be
* exchanged between two processors.
-----
*/
int Test_0(P, T1, T2, i, j, avail_p, w)
set P[MAX_TASKS];
int *T1, *T2, i, j, avail_p,
    w[MAX_TASKS];

{
    *T1 = set_sum(P[i], w);
    *T2 = set_sum(P[j], w);
    if (*T1 > *T2)
        return *T1;
    else
        return *T2;
}

/*
-----
*                                     print_adj_mtx()
-----
* Print the adjacency matrix for independent task system
-----
*/
void print_adj_mtx(n_tasks)
int n_tasks;
{
    int i, j;                /* loop counters */
    FILE *fp;                /* file pointer to adjacency matrix file */

    /*
    * open the adjacency matrix file
    */
    if ((fp = fopen("adj_file", "w")) == NULL) {
        printf("**** Error: cannot open adj_file\n");
        exit(1);
    }

    /*
    * create adjacency matrix for independent task system by placing
    * '0's in all rows.
    */
    for(i=1; i<=n_tasks; i++) {
        for (j=1; j<=i; j++)
            fprintf(fp, " ");
        for(j=i; j<=n_tasks; j++)
            fprintf(fp, "%d", 0);
        fprintf(fp, "\n");
    }
}

```



```

    avail_p,          /* number of processors used */
    LB;               /* Lower bound on schedule */
{
    int i,             /* loop counter */
        smallest;      /* processor id of the minimum sch. length proc.*/

    /*
     * find processor id of a processor whose schedule length is the minimum.
     */
    smallest = 1;
    for (i=2; i<=avail_p; i++)
        if (Proc[i] < Proc[smallest])
            smallest = i;
    return smallest;
}

/*
 *-----
 *                          schedule()
 *-----
 * Divide and Fold algorithm which creates the static schedule for the
 * given task system
 *-----
 */
void schedule(w, Proc , avail_p, LB, n_tasks, P)
int w[MAX_TASKS],    /* weights of tasks in the task system */
    Proc[MAX_PROC],  /* schedule length of each processor */
    avail_p,         /* number of processors used */
    LB,              /* lower bound */
    n_tasks;         /* number of tasks in the task system */
set P[MAX_TASKS];    /* static schedule created */
{
    set S[MAX_TASKS][MAX_TASKS]; /* layers used in D&F */
    int task_id,           /* task id */
        proc_id,          /* processor id */
        n_sets,           /* number of partitions */
        step,             /* number of steps required */
        p_cnt,            /* processor count */
        li_proc,          /* index of current processor */
        i, j,             /* loop counters */
        n_p, set_cnt,
        k, q, odd;

    /*
     * initialize the number tasks assigned to each processor to 0
     */
    for (i=1; i<=avail_p; i++)
        set_init(&P[i]);

    /*
     * initialize schedule length of each processor to 0
     */
    for (i=1; i<=avail_p; i++)
        Proc[i] = 0;

    /*
     * find the number of partitions to be made
     */
    q = ceil((double)n_tasks/avail_p);

    if (is_odd(q)) {
        q = q + 1;
    }

    /*
     * initialize the number of tasks in a partition
     */
    for (i=1; i<=q; i++)
        for (j=1; j<=avail_p; j++)
            set_init(&S[i][j]);

    /*
     * put one task in each partition created
     */
    task_id = 0;
    i = 0;
    while (task_id < n_tasks) {
        i++;
        p_cnt = 0;

```

```

    avail_p,          /* number of processors used */
    LB;               /* Lower bound on schedule */
}
int i,                /* loop counter */
    smallest;         /* processor id of the minimum sch. length proc.*/

/*
 * find processor id of a processor whose schedule length is the minimum.
 */
smallest = 1;
for (i=2; i<=avail_p; i++)
    if (Proc[i] < Proc[smallest])
        smallest = i;
return smallest;
}

/*
 * -----
 *                               schedule()
 * -----
 * Divide and Fold algorithm which creates the static schedule for the
 * given task system
 * -----
 */
void schedule(w, Proc , avail_p, LB, n_tasks, P)
int w[MAX_TASKS],    /* weights of tasks in the task system */
    Proc[MAX_PROC],  /* schedule length of each processor */
    avail_p,         /* number of processors used */
    LB,              /* lower bound */
    n_tasks;         /* number of tasks in the task system */
set P[MAX_TASKS];    /* static schedule created */
{
    set S[MAX_TASKS][MAX_TASKS]; /* layers used in D&F */
    int task_id,             /* task id */
        proc_id,            /* processor id */
        n_sets,             /* number of partitions */
        step,               /* number of steps required */
        p_cnt,              /* processor count */
        li_proc,            /* index of current processor */
        i, j,               /* loop counters */
        n_p, set_cnt,
        k, q, odd;

    /*
     * initialize the number tasks assigned to each processor to 0
     */
    for (i=1; i<=avail_p; i++)
        set_init(&P[i]);

    /*
     * initialize schedule length of each processor to 0
     */
    for (i=1; i<=avail_p; i++)
        Proc[i] = 0;

    /*
     * find the number of partitions to be made
     */
    q = ceil((double)n_tasks/avail_p);

    if (is_odd(q)) {
        q = q + 1;
    }

    /*
     * initialize the number of tasks in a partition
     */
    for (i=1; i<=q; i++)
        for (j=1; j<=avail_p; j++)
            set_init(&S[i][j]);

    /*
     * put one task in each partition created
     */
    task_id = 0;
    i = 0;
    while (task_id < n_tasks) {
        i++;
        p_cnt = 0;

```



```

while ((p_cnt < avail_p) && (task_id < n_tasks)) {
    task_id++;
    p_cnt++;
    set_add(task_id, &S[i][p_cnt]);
}

/*
 * Divide and fold till the number of partitions is equal to avail_p and
 * each partition has tasks/avail_p tasks
 */
if (avail_p > 1) {
    step = n_tasks/2;
    n_sets = q;
    while (step > 0) {
        for (i=1; i<=n_sets/2; i++)
            for (j=1; j<=avail_p; j++) {
                for (k=1; k<=S[n_sets-i+1][avail_p-j+1].size; k++)
                    set_add(S[n_sets-i+1][avail_p-j+1].elem[k], &S[i][j]);
                set_init(&S[n_sets-i+1][avail_p-j+1]);
            }
        n_sets = n_sets/2;
        if (is_odd(n_sets))
            n_sets = n_sets + 1;
        step = step/2;
    }

    /*
     * Assign the number of tasks in each partition to the processors used, i.e.,
     * find the static schedule
     */
    for (i=1; i<=avail_p; i++)
        for (j=1; j<=S[1][i].size; j++)
            set_add(S[1][i].elem[j], &P[i]);

    /*
     * find the schedule length of each processor
     */
    for (i=1; i<=avail_p; i++)
        for (j=1; j<=S[1][i].size; j++)
            Proc[i] = Proc[i] + w[S[1][i].elem[j]];
}

/*
 * if the number of tasks in the task system is less than the number of processors
 * requested, then assign one task to each processor.
 */
else {
    /*
     * create the static schedule
     */
    for (j=1; j<=n_tasks; j++)
        set_add(j, &P[1]);

    /*
     * find the schedule length of each processor
     */
    for (j=1; j<=n_tasks; j++)
        Proc[1] = Proc[1] + w[j];
}

}

/*
-----
 *
 * sort_set()
 *-----
 * Sort elements in a set
 *-----
 */
void sort_set(S, w, n_p)
set S[MAX_TASKS]; /* set to be sorted */
int w[MAX_TASKS], n_p; /* weight of tasks in the task system */
{
    int i, /* loop control variable */
        smallest; /* smallest index */

    /*
     * sort given set
     */
    for (i=n_p; i>=1; i--) {
        smallest = set_index_of_smallest(w, S, i); /* find the smallest elem */

```

```

    set_swap(S, smallest, i); /* swap current element with smallest one */
}
/*
-----
is_odd()
-----
* Checks if an integer is odd.
-----
*/
int is_odd(n)
int n; /* integer to be checked */
{
    if (n%2 > 0) /* check mode of 2 */
        return TRUE;
    else
        return FALSE;
}
/*
-----
print_export_data()
-----
* Create the static schedule in the format required by the scheduler to
* schedule on real processors.
-----
*/
void print_export_data(P, avail_p, n_tasks, w)
set P[MAX_PROC]; /* tasks assigned to a processor */
int avail_p, /* number of processors */
    n_tasks, /* number of tasks in task system */
    w[MAX_TASKS]; /* weight of tasks */
{
    int task_id, /* task id */
        proc_id, /* processor id */
        i, /* loop counter */
        index;
    char no_task[3]; /* used in getting the required output format */

    strcpy(no_task, "-1");
    /*
    * generates the static schedule in the format used by the scheduler
    */
    for (task_id=1; task_id<=n_tasks; task_id++) {
        for (proc_id = 1; proc_id <=avail_p; proc_id++) {
            index = set_find(task_id, &P[proc_id]);
            if (index != 0) {
                printf("%4d", w[task_id]);
                set_del(task_id, &P[proc_id]);
            }
            else
                printf("%4s", no_task);
        }
        printf("\n");
    }
}
/*
-----
calc_LB()
-----
* Calculates the Lower bound for schedule
-----
*/
int calc_LB(w, avail_p, n_tasks)
int w[MAX_TASKS], /* weight of tasks */
    avail_p, /* number of processors used */
    n_tasks; /* number of tasks in task system */
{
    int sum_o_weights, /* sum of weights */
        i, /* loop counter */
        LB; /* Lower bound found */

    /*
    * initialize sum of weights variable
    */
    sum_o_weights = 0;

    /*
    * find sum of lengths of all the tasks in the task system

```

```

    */
    for (i=1; i<=n_tasks; i++)
        sum_o_weights = sum_o_weights + w[i];

    /*
     * finds the lower bound
     */
    if ((sum_o_weights / avail_p) > w[1])
        LB = sum_o_weights / avail_p;
    else
        LB = w[1];
    return LB;
}

/*
-----
 *
 *                               index_of_smallest()
-----
 * Finds a task whose length is the smallest. This procedure is used in the
 * select sort.
-----
 */
int index_of_smallest(w, n)
int w[MAX_TASKS],      /* weights of tasks */
n;                    /* range to be used */
{
    int smallest, i;

    smallest = 1;
    /*
     * finds the smallest length task
     */
    for (i=2; i<=n; i++)
        if (w[i] <= w[smallest])
            smallest = i;
    return smallest;
}

/*
-----
 *
 *                               set_index_of_smallest()
-----
 * find the index of element in set which is smallest.
-----
 */
int set_index_of_smallest(w, S, n)
int w[MAX_TASKS];      /* weights of tasks in the task system */
int S[MAX_TASKS];      /* Set of tasks */
int n;
{
    int smallest,      /* index of the smallest element */
        i;            /* loop counter */

    /*
     * find index of the smallest element
     */
    smallest = 1;
    for (i=2; i<=n; i++)
        if (set_sum(S[i], w) <= set_sum(S[smallest], w))
            smallest = i;
    return smallest;
}

/*
-----
 *
 *                               set_sum()
-----
 * find sum of weights of all the tasks in the set
-----
 */
int set_sum(s, w)
int s;                /* set */
int w[MAX_TASKS];      /* weight of tasks in the task system */
{
    int i,            /* loop counter variable */
        sum;          /* sum of weights */

    /*
     * find sum of weight

```

```

    */
    sum = 0;
    for (i=1; i<=s.size; i++)
        sum = sum + w[s.elem[i]];
    return sum;
}

/*
-----
*
*                                     set_swap()
-----
* swap elements in a set
-----
*/
int set_swap(S, smallest, i)
set S[MAX_TASKS];          /* set */
int smallest,              /* element with the smallest index */
i;                          /* index of an element to be swapped with */
{
    set temp;

    /*
     * swap two elements of a set
     */
    set_copy(S[i], &temp);
    set_copy(S[smallest], &S[i]);
    set_copy(temp, &S[smallest]);
}

/*
-----
*
*                                     swap()
-----
* swaps two elements, used in select sort
-----
*/
int swap(w, smallest, i)
int w[MAX_TASKS],          /* weights of tasks */
smallest,                  /* task id of the smallest lenght task */
i;                          /* index of the task to be swapped with */
{
    int temp;

    /*
     * swaps tasks at index i and smallest
     */
    temp = w[i];
    w[i] = w[smallest];
    w[smallest] = temp;
}

/*
-----
*
*                                     select_sort()
-----
* sorts tasks in the structure w in descending order of length
-----
*/
void select_sort(w, n_tasks)
int w[MAX_TASKS],          /* weight of task */
n_tasks;                  /* number of tasks in task system */
{
    int i,                  /* loop counter */
smallest;                  /* index of smallest lenght task */

    /*
     * does select sort by sorting tasks in descending order of their
     * length
     */
    for (i=n_tasks; i>=1; i--) {
        smallest = index_of_smallest(w, i);
        swap(w, smallest, i);
    }
}

/*
-----
*
*                                     read_weight()
-----
* read weights of tasks in the variable "w" from "task_file"

```

```

-----
*/
void read_weight(w, n_tasks, argv)
int w[MAX_TASKS],          /* weights of tasks */
    *n_tasks;              /* number of tasks in task system */
char *argv[];              /* name of a task system file */
{
    FILE *tfp;              /* pointer to task system file */
    char task_w[4],         /* weight of a task */
        task_file[20];      /* name of task file */
    int task_id;            /* task id */

    /*
     * open a task system file
     */
    strcpy(task_file, argv[1]);
    if ((tfp = fopen(task_file, "r")) == NULL) {
        printf("*** Error: Task file name not given\n");
        exit(1);
    }

    /*
     * read tasks from the task system file in to the structure "w"
     */
    task_id = 0;
    fgets(task_w, MAX_TASKS+1, tfp);
    while (!feof(tfp)) {
        task_id++;
        w[task_id] = atoi(task_w);
        fgets(task_w, MAX_TASKS+1, tfp);
    }
    *n_tasks = task_id;
}

/*=====
 *                               rw.c                               *
 *=====*/
#include "set.h"

#define FALSE      0
#define TRUE       1
#define OUT_FILE   "rw.dat"

/*
 * processing information of task
 */
typedef struct {
    int id,                /* task id */
        start,            /* start time */
        finish;           /* end time */
} task_info;

/*
 * static schedule information about a processor
 */
typedef struct {
    set tasks;             /* tasks assigned to a processor */
    int time_available;    /* time available */
} processor;

void read_adj_mtx();
void read_weight();
void read_layers();
void find_ranked_weight();
void find_task_level();
int min_level();
int task_to_be_scheduled();
int all_Q_empty();
void idle();
void update_available_time();
void assign_proc();
void release_descendants();
void collect_data();
void print_export_data();
void find_critical_path();
int schedule_LB();

```

```

/*
-----
*
*                               main()
*
-----
* In order to run rw.c following command should be given at system prompt
*
* $ rw adj_file task_file layer P rw.sch > out_put
*   where
*       adj_file: file having adjacency matrix
*       task_file: name of file having task system. This is create by
*                   calling task system generator i.e. "indtg". In order
*                   to execute "indtg" following command should be given
*                   at the $ propt. i.e.,
*
*                   $ indtg T min_len max_len > task_file
*                   where
*
*                       T: number of tasks
*                       min_len: minimum lenght of task
*                       max_len: maximum lenght of task
*
*       layer: file having ESP layers for the task system
*       P: number of processors
*       rw.sch: schedule created by ranked weight algorithm
*       out_file: name of an output file
*
* The output of this program are in two files
*   1. rw.sch : have the static schedule created by variant load algorithm
*   2. out_file: have data collected as a result of this static schedule
*   this out_put contain folloing inforamtion about the
*   schedule.
*   1. number of tasks in task system
*   2. number of processors used
*   3. total time taken by the task system
*   4. total processing time
*   5. total idle time
*   6. Lower bound of schedule
*   7. schedule length
*   8. Performance ratio
*   9. Avarage turnaround time
*
-----
*/
main(argc, argv)
int argc;
char *argv[];
{
    char *adj_file,           /* adjacency matrix file name */
          *weight_file;       /* task weight file name */
    set   layer[MAX_TASKS];    /* ESP layers */
    int   n_tasks,            /* number of tasks in the task system */
          LBs,                /* Lower bound */
          n_layers, col;
    int   adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
          w[MAX_TASKS],          /* weight of tasks in task system */
          rw[MAX_TASKS],         /* ranked weight of tasks */
          cp[MAX_TASKS],         /* critical path */
          level[MAX_TASKS],      /* level of a task */
          i, l, j, t, tl,        /* loop counters */
          avail_p,              /* number of processors used */
          p_index, nodes, done,
          task_id;              /* task id */
    set   Q[MAX_TASKS];         /* tasks at particular level */
    processor P[MAX_PROC];      /* static schedule created */
    task_info task_data[MAX_TASKS]; /* task infomation */

    /*
     * make sure that the proper number of parameters have been passed
     */
    if (argc != 6) {
        printf("*** Error: Invalid number of parameters \n");
        exit(1);
    }

    read_adj_mtx(adj_mtx, &n_tasks, argv); /* read adjacency matrix */
    read_weight(w, n_tasks, argv);          /* read weights of tasks */
    read_layers(layer, &n_layers, argv);    /* read layers */

    find_ranked_weight(adj_mtx, w, rw, n_tasks); /* find ranked weights of tasks*/
    find_task_level(layer, level, n_layers);    /* find levels of tasks */
    find_critical_path(cp, w, layer, n_layers, n_tasks); /* find critical path */

    avail_p = atoi(argv[4]);                /* read the number of processors used */
}

```

```

/*
 * initialize the time available at processor to 0
 */
for (i=1; i<= avail_p; i++)
    P[i].time_available = 0;

/*
 * Initialize the number of tasks at each layer to 0
 */
for(i=1; i<=n_layers; i++)
    set_init(&Q[i]);

/*
 * add source node to the first level
 */
set_add(1, &Q[1]);

done = FALSE;
nodes = 0;

while (!done) {
    /*
     * find processor to which task is to be assigned
     */
    p_index = find_processor(P, avail_p);
    /*
     * release descendents of the task that has been scheduled
     */
    release_descendants(P, P[p_index].time_available, avail_p,
                      n_tasks, adj_mtx, Q, level);

    /*
     * find the level in which task is to be inserted
     */
    l = min_level(Q);
    if (l<=n_layers) {
        /*
         * find the task to be scheduled
         */
        t = task_to_be_scheduled(Q[l], rw);

        /*
         * remove a task from predecessor list
         */
        set_del(t, &Q[l]);

        /*
         * assign a task to processor
         */
        assign_proc(t, w[t], p_index, P, task_data);
        nodes++;
        if (Q[l].size == 0) {
            /*
             * update available time at the processor
             */
            update_available_time(P, avail_p, w[t]);
        }
    }
    else if (nodes < n_tasks) {
        /*
         * if no task is available , add idle time
         */
        idle(P, avail_p, p_index);
    }
    else
        done = TRUE;
}

/*
 * find lower bound on schedule
 */
LBs = schedule_LB(w ,rw, cp, n_tasks, avail_p);

/*
 * print the static schedule created in the format required by the
 * scheduler
 */
print_export_data(P, task_data, avail_p, n_tasks, w);
/*

```

```

    * collect performance data
    */
collect_data(P, task_data, avail_p, n_tasks, w, rw, cp, LBs, argv[5]);
}

/*
-----
*
*                                schedule_LB()
*-----
* Find lower bound on schedule lenght
*-----
*/
int schedule_LB(w, rw, cp, n_tasks, avail_p)
int w[MAX_TASKS],          /* weight of tasks in the task system */
    rw[MAX_TASKS],         /* ranked weights of tasks in task system */
    cp[MAX_TASKS],         /* critical path from each task */
    n_tasks,               /* number of tasks in the task system */
    avail_p;               /* number of available processors */
{
    int LB, X1, X2,
        i;

    /*
    * find lower bound
    */
    X1 = (rw[1] - w[1] - w[n_tasks])/avail_p + w[1] + w[n_tasks];
    X2 = X1 / avail_p + 1;
    LB = X2 + (w[1] + w[n_tasks]);

    if (cp[1] > LB)
        LB = cp[1];
    return LB;
}

/*
-----
*
*                                find_critical_path()
*-----
*
*-----
*/
void find_critical_path(cp, w, layer, n_layer, n_tasks)
int cp[MAX_TASKS],         /* critical path from each task */
    w[MAX_TASKS];          /* weights of tasks in the task system */
set layer[MAX_TASKS];      /* layer a task belong to */
int n_layer,               /* number of layers */
    n_tasks;               /* number of tasks */
{
    int task_id,           /* task id */
        level,
        i,
        largest,          /* task with the largest weight */
        ac_cp,            /* accumulated critical path */
        level_task;       /* current level */

    /*
    * find critical path of tasks at the last level
    */
    for (level_task = 1; level_task <= layer[n_layer].size; level_task++) {
        task_id = layer[n_layer].elem[level_task];
        cp[task_id] = w[task_id];
    }

    ac_cp = 0;
    for (level = n_layer-1; level >= 1; level--) {
        /*
        * find the largest task at current level
        */
        largest = w[layer[level+1].elem[1]];
        for (level_task = 2; level_task <= layer[level+1].size; level_task++) {
            task_id = layer[level+1].elem[level_task];
            if (w[task_id] > largest) {
                largest = w[task_id];
            }
        }
        /*
        * add this largest task in current level to critical path
        */
    }
}

```



```

    ac_cp = ac_cp + largest;
    for (level_task = 1; level_task <= layer[level].size; level_task++) {
        task_id = layer[level].elem[level_task];
        cp[task_id] = w[task_id] + ac_cp;
    }
}

/*
-----
*
* print_export_data()
*
* Print the static schedule in the format required by the scheduler
*
-----
*/
void print_export_data(P, task_data, avail_p, n_tasks, w)
processor P[MAX_PROC]; /* static scheduling information of a proc. */
task_info task_data[MAX_TASKS]; /* task information */
int avail_p, /* number of processors avail. */
    n_tasks, /* number of tasks */
    w[MAX_TASKS]; /* weights of tasks */
{
    int task_id, /* task id */
        proc_id, /* processor id */
        index;
    char no_task[3]; /* temp variable */

    strcpy(no_task, "-1");
    for (task_id=1; task_id<=n_tasks; task_id++) {
        /*
        * find the processor assigned to a task
        */
        for (proc_id = 1; proc_id <=avail_p; proc_id++) {
            index = set_find(task_id, &P[proc_id]);
            /*
            * print the task in required format
            */
            if (index != 0) {
                printf("%4d", w[task_id]);
                set_del(task_id, &P[proc_id]);
            }
            else
                printf("%4s", no_task);
        }
        printf("\n");
    }
}

/*
-----
*
* collect_data()
*
* collect performance data for the output
*
-----
*/
void collect_data(P, task_data, avail_p, n_tasks, w, rw, cp, LB, data_file)
processor P[MAX_PROC]; /* static scheduling information of proc. */
task_info task_data[MAX_TASKS]; /* task information */
int avail_p, /* number of available processors */
    n_tasks, /* number of tasks in the task system */
    w[MAX_TASKS], /* weights of task */
    rw[MAX_TASKS], /* ranked weight of task */
    cp[MAX_TASKS], /* critical path */
    LB; /* lower bound on schedule length */
char *data_file; /* output file name */
{
    int i, j, /* loop counters */
        sch_length, /* schedule length */
        idle_tm, /* idle time */
        proc_tm, /* processeing time */
        task_id, /* task id */
        critical_path, /* critical path */
        total_time; /* total time */

    float av_tt, /* average turnaround time */
        pr; /* performance ratio */
    FILE *ofp; /* pointer to the output file */

    /*
    * open an output file
    */

```

```

if ((ofp = fopen(data_file, "a+")) == NULL) {
    printf("**** Error: Cannot open schedule file\n");
    exit(1);
}

/*
 * find schedule length
 */
sch_length = P[1].time_available;
for (i=1; i<=avail_p; i++)
    if (P[i].time_available > sch_length)
        sch_length = P[i].time_available;

/*
 * find critical path
 */
critical_path = cp[1];
for (i=2; i<=n_tasks; i++)
    if (cp[i] > critical_path)
        critical_path = cp[i];

/*
 * find processing time
 */
total_time = sch_length * avail_p;
proc_tm = 0;
for (i=1; i<=n_tasks; i++)
    proc_tm = proc_tm + w[i];

/*
 * find idle time, average turnaround time, and performance ratio
 */
idle_tm = total_time - proc_tm;
av_tt = (float)total_time / (float)n_tasks;
pr = (float) LB / sch_length;

/*
 * print the data collected to output file
 */
fprintf(ofp, "%7d%7d%7d%7d%7d%7d%7d%7.2f%7.2f\n", n_tasks, avail_p,
        total_time, proc_tm, idle_tm, critical_path, LB, sch_length, pr, av_tt);
}

/*
 * -----
 *                               idle()
 * -----
 * Add idle time to those processors which are idle
 * -----
 */
void idle(P, avail_p, p_index)
processor P[MAX_TASKS]; /* static schedule information */
int    avail_p,          /* number of processors */
      p_index;           /* index of processor */
{
    int smallest,          /* smallest time */
        next_smallest, i;

    /*
     * find the smallest time available in all the processors
     */
    next_smallest = P[1].time_available;
    for (i=1; i<=avail_p; i++)
        if (P[i].time_available > next_smallest)
            next_smallest = P[i].time_available;

    smallest = P[p_index].time_available;

    for (i=1; i<= avail_p; i++)
        if ((P[i].time_available < next_smallest) &&
            (P[i].time_available > smallest))
            next_smallest = P[i].time_available;

    /*
     * if available time at a processor is less than the smallest time
     * then add the smallest time
     */
    for (i=1; i<= avail_p; i++)
        if (P[i].time_available < next_smallest)
            P[i].time_available = next_smallest;
}

```

```

/*
-----
*
* update_available_time()
-----
* update available time at all the processors
-----
*/
void update_available_time(P, avail_p, time)
processor P[MAX_PROC]; /* static schedule information of processors */
int      avail_p,      /* number of processors */
      time;            /* time to be added for available time */
{
    int index;          /* index of processor */

    /*
     * update available time of all processors
     */
    for (index = 1; index <= avail_p; index++)
        if (P[index].tasks.size == 0)
            P[index].time_available = P[index].time_available + time;
}

/*
-----
*
* assign_proc()
-----
* assigne a processor to a task
-----
*/
void assign_proc(task_id, weight, p_index, P, task_data)
int task_id,          /* task id */
    weight,           /* weight of task */
    p_index;          /* processor index */
processor P[MAX_PROC]; /* static schedule information */
task_info task_data[MAX_TASKS]; /* task information */
{
    task_data[task_id].id = task_id;

    /*
     * get processing information of task
     */
    task_data[task_id].start = P[p_index].time_available;
    task_data[task_id].finish = task_data[task_id].start + weight;
    P[p_index].time_available = task_data[task_id].finish;

    /*
     * add task to the processor task list
     */
    set_add(task_id, &P[p_index]);
}

/*
-----
*
* release_descendants()
-----
* Release descendents of a task when the execution of task is completed.
-----
*/
void release_descendants(P, time, avail_p, n_tasks, adj_mtx, Q, level)
processor P[MAX_PROC]; /* static schedule information */
int      time,         /* time task completed execution */
      avail_p,        /* number of processors */
      n_tasks,        /* number of tasks */
      adj_mtx[MAX_TASKS][MAX_TASKS]; /* adjacency matrix */
set      Q[MAX_TASKS]; /* levels */
int      level[MAX_TASKS]; /* level information of task */
{
    int id,            /* task id */
        i, j,         /* loop counters */
        tail;         /* index of task at the tail */

    for (i=1; i<=avail_p; i++) {
        if ((P[i].time_available == time) && (P[i].tasks.size != 0)) {
            /*
             * find task at the tail of the processor
             */
            tail = P[i].tasks.size;
            id = P[i].tasks.elem[tail];

```

```

/*
 * remove current task as predecessor of tasks in the task system
 */
for (j=id; j<=n_tasks; j++) {
    if ( adj_mtx[id][j] == 1) {
        adj_mtx[id][j] = 0;
        if (pred_found(adj_mtx, j) == FALSE) {
            set_add(j, &Q[level[j]]);
        }
    }
}
}
}

/*
-----
 *
 * find_processor()
-----
 * find the id of processor to which a task could be assigned
-----
 */
int find_processor(P, avail_p)
processor P[MAX_PROC];          /* static schedule of processor */
int avail_p;                   /* number of processors */
{
    int i, time,
        find_proc;              /* id of found processor */

    find_proc = 1;
    time = P[1].time_available;
    i = 2;
    while (i <= avail_p) {
        /*
         * find processor whose available time is less than current time
         */
        if (P[i].time_available < time) {
            time = P[i].time_available;
            find_proc = i;
        }
        i++;
    }
    return find_proc;
}

/*
-----
 *
 * all_Q_empty()
-----
 * check if all the layers are empty
-----
 */
int all_Q_empty(Q, n_layers)
set Q[MAX_TASKS];              /* layers */
int n_layers;                  /* number of layers */
{
    int i,
        empty;                  /* loop counter */
                                /* if TRUE, all layers are empty */

    empty = TRUE;
    for (i=1; i<=n_layers; i++)
        if (Q[i].size != 0)
            empty = FALSE;
    return empty;
}

/*
-----
 *
 * pred_found()
-----
 * check if a task has predecessors
-----
 */
int pred_found(adj_mtx, t1)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
t1; /* task id */
{
    int i,
        pred;                   /* loop counter */
                                /* if TRUE, it has predecessors */

    pred = FALSE;

```



```

{
FILE *tfp;                                /* adjacency matrix file pointer */
char instr[MAX_TASKS+1],
    sch[3],
    adj_file[20];                          /* name of adjacency matrix file */
int task_cnt,                             /* task count */
    succ_cnt;                             /* successor count */

/*
 * open adjacency matrix file
 */
strcpy(adj_file, argv[1]);
if ((tfp = fopen(adj_file, "r")) == NULL) {
    printf("*** Error: adjacency file name not given\n");
    exit(1);
}

task_cnt = 0;
fgets(instr, MAX_TASKS+1, tfp); /* read one row of the adjacency matrix */
while (!feof(tfp)) {
    task_cnt++;
    succ_cnt = 1;
    /*
     * find number of successors of the current task
     */
    while (instr[succ_cnt] == ' ') succ_cnt++;
    /*
     * make row of adjacency matrix
     */
    while (instr[succ_cnt] != '\0') {
        sch[0] = instr[succ_cnt-1];
        sch[1] = '\0';
        adj_mtx[task_cnt][succ_cnt] = atoi(sch);
        succ_cnt++;
    }
    fgets(instr, MAX_TASKS+1, tfp); /* get next row */
}
*n_tasks = task_cnt;
}

/*
-----
*
* read_layers()
-----
* read ESP layers form the layer file
-----
*/
void read_layers(layers, n_layers, argv)
set layers[MAX_TASKS]; /* layers set */
int *n_layers;          /* number of layers */
char *argv[];           /* file having layers */
{
FILE *tfp;              /* file pointer to layer file */
char instr[MAX_TASKS+1],
    sch[3],
    layer_file[20],     /* layer file name */
    *p;
int layer_cnt,          /* number of layers */
    elem_cnt;           /* number of tasks in a layer */

/*
 * open layer file
 */
strcpy(layer_file, argv[3]);
if ((tfp = fopen(layer_file, "r")) == NULL) {
    printf("*** Error: layer file name not given\n");
    exit(1);
}

layer_cnt = 0;
/*
 * read the first layer
 */
fgets(instr, MAX_TASKS+1, tfp);
while (!feof(tfp)) {
    layer_cnt++; /* increment the layer count */
    p = strtok(instr, " ");
    set_init(&layers[layer_cnt]); /* initialize current layer */
    /*
     * add tasks in layer to "layers" structure
     */
}
}

```

```

    set_add(atoi(p), &layers[layer_cnt]);
    do {
        p = strtok('\0', " ");
        if (p)
            set_add(atoi(p), &layers[layer_cnt]);
    } while (p);
    fgets(instr, MAX_TASKS+1, tfp); /* read next layer */
}
*n_layers = layer_cnt;
}
/*
-----
*
-----
* read weights of tasks from task file
-----
*/
void read_weight(w, n_tasks, argv)
int w[MAX_TASKS], /* weights of task */
    n_tasks; /* number of tasks */
char *argv[]; /* file having weights of tasks */
{
    FILE *tfp; /* file pointer to the task file */
    char task_w[4], /* weight of a task */
        task_file[20]; /* task file name */
    int task_id; /* task id */

    /*
    * open task file
    */
    strcpy(task_file, argv[2]);
    if ((tfp = fopen(task_file, "r")) == NULL) {
        printf("**** Error: task file name not given\n");
        exit(1);
    }

    /*
    * read task weight from task file
    */
    for (task_id = 1; task_id <= n_tasks; task_id++) {
        fgets(task_w, MAX_TASKS+1, tfp);
        w[task_id] = atoi(task_w);
    }
}

/*
-----
*
-----
* find ranked weight of tasks in the task system
-----
*/
void find_ranked_weight(adj_mtx, w, rw, n_tasks)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    w[MAX_TASKS], /* weights of tasks */
    rw[MAX_TASKS], /* ranked weights of tasks */
    n_tasks; /* number of tasks in the task system */
{
    int row, /* current row */
        col, /* current column */
        task_cnt, /* task count */
        i, j; /* loop counters */
    set_succ[MAX_TASKS]; /* successors to a task */

    /*
    * initialize successor list to 0
    */
    for (i=1; i<=n_tasks; i++)
        set_init(&succ[i]);

    /*
    * ranked-weight of last task is equal to the weight of task
    */
    rw[n_tasks] = w[n_tasks];

    /*
    * find all the successors to a task
    */
    for (row=n_tasks-1; row>=1; row--)
        for (col=row+1; col<=n_tasks; col++)

```

```

        if (adj_mtx[row][col] == 1) {
            set_add(col, &succ[row]);
            set_union(&succ[row], succ[col]);
        }
    }

    /*
     * ranked weight is the sum of weights of all the successors of
     * a task
     */

    for (i=1; i<=n_tasks; i++) {
        rw[i] = w[i];
        for (j=1; j<=succ[i].size; j++)
            rw[i] = rw[i] + w[succ[i].elem[j]];
    }
}

/*=====
 *                               vl_esp.c                               *
 *=====*/
#include "set.h"

#define FALSE      0
#define TRUE       1
#define OUT_FILE   "vl_esp.sch"

void read_weight();
void read_adj_mtx();
void select_sort();
int  calc_LB();
void schedule();
void print_export_data();
void collect_data();
void read_layers();
void idle();
int  schedule_LB();

/*
 * -----
 *                               main()
 * -----
 * In order to run vl_esp.c following command should be given at system prompt
 *
 * $ vl_esp adj_file task_file layer P vl_esp.sch > out_put
 *   where
 *       adj_file: file having adjacency matrix
 *       task_file: name of file having task system. This is create by
 *                   calling the task system generator i.e. "indtg". In order
 *                   to execute "indtg" following command should be given
 *                   at the $ propt. i.e.,
 *
 *                   $ indtg T min_len max_len > task_file
 *                   where
 *
 *                       T: number of tasks
 *                   min_len: minimum length of task
 *                   max_len: maximum length of task
 *
 *       layer: file having ESP layers for the task system
 *       P: number of processors
 *       vl_esp.sch: schedule created by ranked weight algorithm
 *       out_file: name of output file
 *
 * The outputs of this program are in two files
 * 1. vl_esp.sch : have the static schedule created by variant load algorithm
 * 2. out_file: have data collected as a result of this static schedule
 *               this out_put contain follwoing inforamtion about the
 *               schedule.
 *               1. number of tasks in task system
 *               2. number of processors used
 *               3. total time taken by the task system
 *               4. total processing time
 *               5. total idle time
 *               6. Lower bound of schedule
 *               7. schedule length
 *               8. Performance ratio
 *               9. Avarage turnaround time
 * -----
 */

main(argc, argv)
int argc;

```



```

char *argv[];
{
    int  n_tasks,           /* number of tasks */
        nl_tasks,         /* number of tasks in each layer */
        n_layers,         /* number of layers */
        avail_p,          /* number of processors */
        LB, LBs,          /* lower bounds */
        i, k,              /* loop counters */
        w[MAX_TASKS],      /* weights of tasks */
        w_index[MAX_TASKS], /* index of a task in layers */
        w_layer[MAX_TASKS], /* weights of tasks in layers */
        Proc[MAX_PROC];    /* schedule lengths of processors */
    set  P[MAX_PROC],      /* static schedule */
        adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
        layers[MAX_TASKS]; /* layers */

    /*
     * check for the correct number of parameters
     */
    if (argc != 6) {
        printf("*** Error: Invalid number of parameters \n");
        exit(1);
    }

    /*
     * initialize the static schedule
     */
    avail_p = atoi(argv[4]);
    for (i=1; i<=avail_p; i++)
        set_init(&P[i]);

    /*
     * read weight of task, adjacency matrix and ESP layer
     */
    read_weight(w, &n_tasks, argv);
    read_adj_mtx(adj_mtx, &n_tasks, argv);
    read_layers(layers, &n_layers, argv);

    /*
     * initialize the schedule length of each processor
     */
    for (i=1; i<=avail_p; i++)
        Proc[i] = 0;

    /*
     * for each of the layer schedule the tasks using variant-load
     * algorithm
     */
    for (k=1; k<=n_layers; k++) {
        nl_tasks = layers[k].size;
        for (i=1; i<=nl_tasks; i++) {
            w_index[i] = layers[k].elem[i];
            w_layer[i] = w[layers[k].elem[i]];
        }
        select_sort(w_index, w_layer, nl_tasks);
        LB = calc_LB(w_layer, avail_p, nl_tasks);
        schedule(w_index, w_layer, Proc, avail_p, LB, nl_tasks, P);
        if (k != n_layers)
            idle(Proc, avail_p);
    }

    /*
     * find the lower bound of the schedule
     */
    LBs = schedule_LB(w, n_tasks, avail_p);

    /*
     * collect the performance data
     */
    collect_data(Proc, avail_p, n_tasks, w, LBs, argv[5]);

    /*
     * create the static schedule in the format required by
     * the scheduler
     */
    print_export_data(P, avail_p, n_tasks, w);
}

/*
 * -----
 *
 *                               schedule_LB()

```

```

*-----
* find the Lower bound of the schedule
*-----
*/
int schedule_LB(w, n_tasks, avail_p)
int w[MAX_TASKS], /* weights of tasks */
    n_tasks,      /* number of tasks */
    avail_p;      /* number of processors */
{
    int seq_time, /* sequential time */
        LB,      /* lower bound */
        i;       /* loop counter */

    seq_time = 0;

    /*
     * find sequential execution time of the task system
     */
    for (i=1; i<=n_tasks; i++)
        seq_time = seq_time + w[i];

    /*
     * find the lower bound
     */
    LB = (seq_time - w[1] - w[n_tasks])/avail_p + w[1] + w[n_tasks];
    return LB;
}

/*
*-----
* read_adj_mtx()
*-----
* read the adjacency matrix from the adjacency matrix file
*-----
*/
void read_adj_mtx(adj_mtx, n_tasks, argv)
int adj_mtx[MAX_TASKS][MAX_TASKS], /* adjacency matrix */
    n_tasks; /* number of tasks */
char *argv[]; /* file having adjacency matrix */
{
    FILE *tfp; /* adjacency matrix file pointer */
    char instr[MAX_TASKS+1],
        sch[3],
        adj_file[20]; /* name of the adjacency matrix file */
    int task_cnt, /* task count */
        succ_cnt; /* successor count */

    /*
     * open adjacency matrix file
     */
    strcpy(adj_file, argv[1]);
    if ((tfp = fopen(adj_file, "r")) == NULL) {
        printf("**** Error: adjecency file name not given\n");
        exit(1);
    }

    task_cnt = 0;
    fgets(instr, MAX_TASKS+1, tfp); /* read one row of the adjacency matrix */
    while (!feof(tfp)) {
        task_cnt++;
        succ_cnt = 1;
        /*
         * find the number of successors of the current task
         */
        while (instr[succ_cnt] == ' ') succ_cnt++;
        /*
         * make row of adjacency matrix
         */
        while (instr[succ_cnt] != '\0') {
            sch[0] = instr[succ_cnt-1];
            sch[1] = '\0';
            adj_mtx[task_cnt][succ_cnt] = atoi(sch);
            succ_cnt++;
        }
        fgets(instr, MAX_TASKS+1, tfp); /* get next row */
    }
    *n_tasks = task_cnt;
}

/*

```

```

-----
/*
-----
idle()
-----
*/
/* add idle time to each of the processor
-----
*/
void idle(Proc, avail_p)
int Proc[MAX_PROC], /* schedule length */
    avail_p; /* number of processor */
{
    int i, l; /* loop counters */

    /*
     * find processor whose available time is largest
     */
    l = index_of_largest(Proc, avail_p);

    /*
     * add idle time
     */
    for (i=1; i<=avail_p; i++)
        Proc[i] = Proc[l];
}

/*
-----
/*
-----
index_of_largest()
-----
/* find processor which has the largest schedule length
-----
*/
int index_of_largest(Proc, avail_p)
int Proc[MAX_PROC], /* schedule length of processors */
    avail_p; /* number of processors */
{
    int i, /* loop counter */
        largest; /* index of the largest schedule length processor */

    largest = 1;

    /*
     * find the processor with the largest schedule length
     */
    for (i=2; i<=avail_p; i++)
        if (Proc[i] >= Proc[largest])
            largest = i;

    return largest;
}

/*
-----
/*
-----
read_layers()
-----
/* read ESP layers form the layer file
-----
*/
void read_layers(layers, n_layers, argv)
set layers[MAX_TASKS]; /* layers set */
int *n_layers; /* number of layers */
char *argv[]; /* file having layers */
{
    FILE *tfp; /* file pointer to layer file */
    char instr[MAX_TASKS+1],
        sch[3],
        layer_file[20], /* layer file name */
        *p;
    int layer_cnt, /* number of layers */
        elem_cnt; /* number of tasks in a layer */

    /*
     * open a layer file
     */
    strcpy(layer_file, argv[3]);
    if ((tfp = fopen(layer_file, "r")) == NULL) {
        printf("*** Error: layer file name not given\n");
        exit(1);
    }

    layer_cnt = 0;

```

```

/*
 * read the first layer
 */
fgets(instr, MAX_TASKS+1, tfp);
while (!feof(tfp)) {
    layer_cnt++; /* increment the layer count */
    p = strtok(instr, " ");
    set_init(&layers[layer_cnt]); /* initialize the current layer */
    /*
     * add tasks in layer to "layers" structure
     */
    set_add(atoi(p), &layers[layer_cnt]);
    do {
        p = strtok('\0', " ");
        if (p)
            set_add(atoi(p), &layers[layer_cnt]);
    } while (p);
    fgets(instr, MAX_TASKS+1, tfp); /* read next layer */
}
*n_layers = layer_cnt;
}

/*
 * -----
 * collect_data()
 * -----
 * collect performance data for the output
 * -----
 */
void collect_data(Proc, avail_p, n_tasks, w, LB, data_file)
int Proc[MAX_PROC]; /* schedule length of processors */
int avail_p, /* number of processors */
    n_tasks, /* number of tasks */
    w[MAX_TASKS], /* weights of tasks */
    LB; /* lower bound */
char *data_file; /* name of the output file */
{
    int i, j, /* loop counter */
        sch_length, /* schedule length */
        idle_tm, /* idle time */
        proc_tm, /* proc time */
        task_id, /* task id */
        total_time; /* total time */

    float av_tt, /* average turnaround time */
          pr; /* performance ratio */
    FILE *ofp; /* pointer to the output file */

    /*
     * open an output file
     */
    if ((ofp = fopen(data_file, "a+")) == NULL) {
        printf("*** Error: cannot open schedule file\n");
        exit(1);
    }
    /*
     * find the schedule length
     */
    sch_length = Proc[1];
    for (i=1; i<=avail_p; i++)
        if (Proc[i] > sch_length)
            sch_length = Proc[i];

    /*
     * find the processing time
     */
    total_time = sch_length * avail_p;
    proc_tm = 0;
    for (i=1; i<=n_tasks; i++)
        proc_tm = proc_tm + w[i];

    /*
     * find idle time , average turnaround time and performance ratio
     */
    idle_tm = total_time - proc_tm;
    av_tt = (float)total_time / (float)n_tasks;
    pr = (float) LB/sch_length;

    /*
     * print the performance data collected to output file

```



```

/*
 * If still some tasks remained to be assigned, find the processor
 * with lowest schedule length and assign tasks to it.
 */
if (task_id <= n_tasks) {
    if (proc_id == avail_p) {
        li_proc = find_lowest_index(Proc, avail_p, LB);
        set_add(w_index[task_id], &P[li_proc]);
        Proc[li_proc] = Proc[li_proc] + w[task_id];
        task_id++;
        if (task_id < n_tasks)
            proc_id = 1;
    }
    else
        proc_id++;
}
}

/*
-----
 *
-----
 * Creates static schedule in the format required by the scheduler to schedule on * real
processors.
-----
 */
void print_export_data(P, avail_p, n_tasks, w)
set P[MAX_PROC];          /* tasks assigned to a processor */
int    avail_p,           /* number of processors */
      n_tasks,            /* number of tasks in task system */
      w[MAX_TASKS];       /* weight of tasks */
{
    int task_id,          /* task id */
        proc_id,         /* processor id */
        i,               /* loop counter */
        index;
    char no_task[3];      /* used in getting required output format */

    strcpy(no_task, "-1");
    /*
     * generates the static schedule in the format used by the scheduler
     */
    for (task_id=1; task_id<=n_tasks; task_id++) {
        for (proc_id = 1; proc_id <=avail_p; proc_id++) {
            index = set_find(task_id, &P[proc_id]);
            if (index != 0) {
                printf("%4d", w[task_id]);
                set_del(task_id, &P[proc_id]);
            }
            else
                printf("%4s", no_task);
        }
        printf("\n");
    }
}

/*
-----
 *
-----
 * Calculates the Lower bound for schedule
-----
 */
int calc_LB(w, avail_p, n_tasks)
int w[MAX_TASKS],          /* weight of tasks */
    avail_p,              /* number of processors used */
    n_tasks;              /* number of tasks in task system */
{
    int sum_o_weights,     /* sum of weights */
        i,               /* loop counter */
        LB;              /* Lower bound found */

    /*
     * initialize sum of weight variable
     */
    sum_o_weights = 0;

    /*
     * find sum of length of all the tasks in the task system

```

```

    */
    for (i=1; i<=n_tasks; i++)
        sum_o_weights = sum_o_weights + w[i];

    /*
     * finds the lower bound
     */
    if ((sum_o_weights / avail_p) > w[1])
        LB = sum_o_weights / avail_p;
    else
        LB = w[1];
    return LB;
}

/*
-----
 *                                     index_of_smallest()
-----
 * Finds a task whose length is smallest. This procedure is used in the
 * select sort.
-----
 */
int index_of_smallest(w, n)
int w[MAX_TASKS],          /* weight of tasks */
n;                          /* range to be used */
{
    int smallest, i;

    smallest = 1;
    /*
     * find task with the smallest length
     */
    for (i=2; i<=n; i++)
        if (w[i] <= w[smallest])
            smallest = i;
    return smallest;
}

/*
-----
 *                                     swap()
-----
 * swaps two elements, used in select sort
-----
 */
int swap(w_index, w_layer, smallest, i)
int w_index[MAX_TASKS],    /* index of tasks in layer */
w_layer[MAX_TASKS],        /* weight of tasks in a layer */
smallest,                  /* task id of the smallest weight task */
i;                          /* index of the task to be swaped with */
{
    int temp;

    /*
     * swap the task id
     */
    temp = w_layer[i];
    w_layer[i] = w_layer[smallest];
    w_layer[smallest] = temp;

    /*
     * swap the task weight
     */
    temp = w_index[i];
    w_index[i] = w_index[smallest];
    w_index[smallest] = temp;
}

/*
-----
 *                                     select_sort()
-----
 * sorts tasks in the structure w in descending order of length
-----
 */
void select_sort(w_index, w_layer, n_tasks)
int w_index[MAX_TASKS],    /* task in a layer */
w_layer[MAX_TASKS],        /* weight of task in a layer */
n_tasks;                  /* number of tasks */

```

```

{
    int i,                /* loop counter */
        smallest;        /* index of the smallest lenght task */

    /*
     * does select sort by sorting tasks in descending order of their
     * lengths
     */
    for (i=n_tasks; i>=1; i--) {
        smallest = index_of_smallest(w_layer, i);
        swap(w_index, w_layer, smallest, i);
    }
}

/*
-----
*
* read_weight()
-----
* reads weight of tasks in the variable "w" from "task_file"
-----
*/
void read_weight(w, n_tasks, argv)
int w[MAX_TASKS],        /* weight of tasks */
    *n_tasks;            /* number of tasks in task system */
char *argv[];            /* name of the task system file */
{
    FILE *tfp;            /* pointer to the task system file */
    char task_w[4],        /* weight of a task */
        task_file[20];    /* name of a task file */
    int task_id;          /* task id */

    /*
     * open task system file
     */
    strcpy(task_file, argv[1]);
    if ((tfp = fopen(task_file, "r")) == NULL) {
        printf("*** Error: Task file name not given\n");
        exit(1);
    }

    /*
     * reads tasks from the task system file in to the structure "w"
     */
    task_id = 0;
    fgets(task_w, MAX_TASKS+1, tfp);
    while (!feof(tfp)) {
        task_id++;
        w[task_id] = atoi(task_w);
        fgets(task_w, MAX_TASKS+1, tfp);
    }
    *n_tasks = task_id;
}

/*=====*/
/*
ss_dep.c
/*=====*/
#include <stdio.h>          /* standard I/O header */
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <parallel/microtask.h> /* microtasking header */
#include <parallel/parallel.h> /* parallel library */

#define SIZE 150            /* size of metrix */
#define UNIT_FILE "unit_file" /* name of the file used to have unit */
/* lenght */
#define MAX_DEP 150        /* Maximum number of dependencies allowed */
#define MAX_PROCESSORS 24 /* Maximum number of processors avail. */
#define MAX_TASK_PJ 150    /* Maximum number of tasks per processor */
#define FALSE 0
#define TRUE 1
#define DONE 0
#define SYSTEM_COMMAND "system_task" /* Task - System command */
#define USER_PROGRAM "user_task" /* Task - used command */
#define WHILE_CT 800000 /* While count for unit time */
#define a 16807.0        /* Variables used by random number */
#define m 2147483647.0    /* generator */
#define q 127773.0        /*
#define r 2836.0          */

```



```

/*
 * It contains general information about the task system to be scheduled,
 * i.e., number of processors used, number of tasks, scheduling algorithm
 * used etc.
 */
struct IN_SCH_INFO {
    int  avail_procs,      /* Number of processors available in the system */
        nprocs,           /* Number of processors requested */
        tasks;            /* Number of tasks in the task system */
    char wait_type,        /* User programs or System command */
        in_out,           /* Run the task inside the scheduler or outside */
        alg[10];          /* scheduling algorithms to be used. */
};

/*
 * Contains processing information about each task in the task system, this
 * includes time at which the processing of a task started and the time at
 * which the processing of a task ended.
 */
struct PROGRESS_DATA {
    int  task_id,          /* task id of a task */
        start_time,       /* time at which processing of a task started */
        end_time;         /* time at which processing of a task ended */
} progress_data;

/*
 * Contain general information about a task, i.e., the processor to which
 * the task is assigned, length of task, type etc.
 */
struct TASK {
    int  processor_id,     /* processor id of the assigned processor */
        length;           /* length of a task */
    char type,             /* User program or System command */
        in_out,           /* Run the task inside the scheduler or outside */
        task[20];         /* name of the task to be executed */
} stat;

/*
 * Contain schedule information i.e., when a task is assigned to a processor
 * then what is the processor id, task length and task id.
 */
struct SCHE {
    int  processor,        /* processor id */
        length,           /* length of the task assigned to above processor */
        task;             /* task id of the task assigned to the above processor */
} sche_info;

/*
 * Contain master clock and start time and end time information of all the
 * tasks in the task system.
 */
struct MISC {
    long clock;            /* Master clock */
    int  clock_stop;       /* if TRUE, the clock is stopped */
    struct PROGRESS_DATA progress[SIZE+1]; /* progress of each task */
} misc;

shared struct SCHE gl_s_info[200]; /* schedule information of tasks */
shared struct TASK gl_task_info[SIZE+1]; /* task information of tasks */
shared int gl_sch_cnt, /* number of tasks scheduled so far */
        gl_sch_list[MAX_PROCESSORS*SIZE+1], /* used for static scheduling */
        gl_adj_mtx[SIZE+1][SIZE+1]; /* adjacency matrix */
shared struct MISC gl_misc_info; /* progress information of tasks */

void initialization();
void init_task_info();
void master_clock();
void itoa();
void read_tasks();
void general_data_ud();
void task();
void task_schedual();
void m_fork();
void m_kill_procs();
void remove_dep();
int  pred_done();
int  find_unit_time();
double random();

```

```

/*
-----
main()
-----
* In order to run the scheduler the following command should be given at
* the $ prompt
* $ ss_dep sch_file adj_file T P Wait_type in_out sch_alg out_file
* where
*   sch_file: file which contains the schedule which is to be used for the
*             scheduling of tasks to the processors
*   adj_file: file which contains the adjacency matrix
*   T: Number of tasks in the task system
*   P: Number of processors to be used for scheduling
*   Wait_type: Wait type i.e., User program or System command.
*               Wait_type = 'W' ( for user command )
*               Wait_type = 'S' ( for System command )
*   in_out: the tasks are to be executed inside or outside the scheduler
*            in_out = 'I' ( for inside the scheduler )
*            in_out = 'O' ( for outside the scheduler )
*   sch_alg: Abreviation of scheduling algorithm to be used
*            sch_alg = 'vl' (for variant-load)
*            sch_alg = 'daf' (for divide & fold )
*            sch_alg = 'rw' ( for ranked weight )
*            sch_alg = 'vl_esp' (for ESP/VL)
*   out_file: Name of the file in which the result should be gone
*-----
*/
main (argc, argv)
int argc;
char *argv[];
{
    struct IN_SCH_INFO ma_in_sch_info; /* task system information */
    FILE *ma_ofp; /* pointer to output file */
    char out_file[40]; /* output file name */
    double seed, /* seed used by the random number generator */
           primer; /* variable used by the random number gen. */
    int i; /* loop counter */

    /*
     * check for correct number of parameters
     */
    if (argc!=9) {
        printf("*** Error: Invalid number of parameters\n");
        exit(1);
    }

    /*
     * copy the input parameters into the ma_in_sch_info structure for later
     * use
     */
    ma_in_sch_info.tasks = atoi(argv[3]);
    ma_in_sch_info.nprocs = atoi(argv[4]);
    ma_in_sch_info.wait_type = argv[5][0];
    ma_in_sch_info.in_out = argv[6][0];
    strcpy(ma_in_sch_info.alg, argv[7]);
    strcpy(out_file, argv[8]);

    /*
     * Initialize the random number generator
     */
    seed = 2.009;
    for (i=1; i<=100; i++)
        primer = random(&seed);

    /*
     * open the output file
     */
    ma_ofp = fopen(out_file, "a+");

    /*
     * call read_tasks() to read schedule information, task information and
     * dependency information (for the dependent task system) for the files
     * specified in the input parameters
     */
    read_tasks(gl_adj_mtx, gl_task_info, &ma_in_sch_info, &ma_ofp,
              gl_sch_list, argv[1], argv[2], &seed);

    /*
     * Initialize data structures
     */
    initialization(gl_task_info, &gl_sch_cnt, &gl_misc_info);
}

```

```

/*
 * tells the operating system number of processors needed for the scheduling
 * of the task system, it should be one more than the number of processors
 * specified at the input parameter, i.e., one for master clock
 */
m_set_procs(ma_in_sch_info.nprocs+1);

/*
 * Call m_fork() to get processors and also specify the procedure
 * that is to be executed by each processor. In this case all the
 * processors are gone executed procedure task_scheduled(). In the fork
 * command we also pass all the shared memory variables used by the
 * processors.
 */
m_fork(task_scheduled, ma_in_sch_info.tasks, gl_task_info, &gl_misc_info,
      &gl_sch_cnt, gl_s_info, gl_sch_list, gl_adj_mtx);

/*
 * Return all the processors to the operating system
 */
m_kill_procs();

/*
 * put the gathered data into the output file
 */
general_data_ud(ma_ofp, gl_misc_info, ma_in_sch_info, gl_sch_cnt, gl_s_info);
}

/*
-----
 *                                     general_data_ud()
-----
 * Output following data for each task in the task system to the output file
 * 1. task id
 * 2. lenght of task
 * 3. processor task was executed on
 * 4. time at which task was assigned to the processor
 * 5. time at which processing of task was completed
 * 6. processing time of the task
-----
 */
void general_data_ud(ofp, misc_info, in_sch_info, sch_cnt, s_info)
FILE *ofp; /* output file pointer */
struct MISC misc_info; /* processing information */
struct IN_SCH_INFO in_sch_info; /* task system information */
int sch_cnt; /* number of tasks scheduled */
struct SCHE s_info[200]; /* schedule information */
{
    int i; /* counter */
    time_diff; /* time difference */
    unit_time; /* unit time */
    nproc; /* number of processors */
    ntasks; /* number of tasks in the task system */
    total_tm; /* total time */
    sch_length; /* schedule length */
    idle_tm; /* idle time */
    proc_tm; /* processing time */
    att; /* average turnaround time */

    FILE *ufp; /* file pointer for unit time file */

    /*
     * open the unit file for read/write
     */
    if ((ufp = fopen(UNIT_FILE, "a+")) == NULL) {
        printf("*** Error: can't open the unit data file ***\n");
        exit(1);
    }

    /*
     * find unit time by calling find_unit_time()
     */
    unit_time = find_unit_time(sch_cnt, misc_info, s_info);

    fprintf(ufp, "%d\n", unit_time); /* put unit time in unit file */

    /*
     * store number of tasks and processors from the structue containing
     * task system information into local variables nproc and ntasks.
     */
    nproc = in_sch_info.nprocs;

```

```

ntasks = in_sch_info.tasks;

/*
 * store schedule length and total time from the structure containing
 * progress information into local variables sch_length and total_tm.
 */
sch_length = misc_info.progress[s_info[sch_cnt].task].end_time;
total_tm = nproc * misc_info.progress[s_info[sch_cnt].task].end_time;

proc_tm = 0;          /* initialize processing time to '0' */

/*
 * calculate total processing time
 */
for (i=1; i<=sch_cnt; i++) {
    time_diff = misc_info.progress[s_info[i].task].end_time -
                misc_info.progress[s_info[i].task].start_time;
    proc_tm = proc_tm + time_diff;
}

/*
 * calculate the idle time and the average turnaround time
 */
idle_tm = total_tm - proc_tm;
att = total_tm / ntasks;

/*
 * calculate the processing time of each task and then
 * output the following results into the output file
 * 1. s_info[i].task           = task id
 * 2. s_info[i].length        = length of task
 * 3. s_info[i].processor      = processor id
 * 4. misc_info.progress[s_info[i].task].start_time = start time
 * 5. misc_info.progress[s_info[i].task].end_time   = end time
 * 6. time_diff               = processing time
 */
for (i=1; i<=sch_cnt; i++) {
    time_diff = misc_info.progress[s_info[i].task].end_time -
                misc_info.progress[s_info[i].task].start_time;
    fprintf(ofp, "%5d %5d %5d %5d %10d %10d\n", i, s_info[i].task,
                s_info[i].length, s_info[i].processor,
                misc_info.progress[s_info[i].task].start_time,
                misc_info.progress[s_info[i].task].end_time,
                time_diff);
}
}

/*
-----
 *
 * find_unit_time()
-----
 * find the unit time, i.e., the number of master clock count for the unit
 * time, this is found by finding the task whose length is minimum and then
 * dividing the clock count difference between the start and the end of
 * execution of task by the length of the task.
-----
 */
int find_unit_time(sch_cnt, misc_info, s_info)
int sch_cnt;          /* number of tasks scheduled */
struct MISC misc_info; /* progress information */
struct SCHE s_info[200]; /* schedule information */
{
    int i,              /* loop counter */
        min,           /* minimum length found */
        min_i,         /* task id of minimum length task */
        unit_time,     /* unit time found */
        time_diff;     /* time difference */

    /*
     * find the task whose length is minimum and also its task id.
     */
    min = 9999;
    for (i=1; i<=sch_cnt; i++) {
        if (s_info[i].length < min) {
            min = s_info[i].length;
            min_i = i;
        }
    }
}

```

```

/*
 * find the unit length by subtracting the end time from the start time
 * for that task and then dividing by the length of the task
 */
time_diff = misc_info.progress[s_info[min_i].task].end_time -
             misc_info.progress[s_info[min_i].task].start_time;
unit_time = time_diff / s_info[min_i].length;

return unit_time; /* return unit time found */
}

/*
-----
 *
-----
 *
-----
 */
void read_tasks(adj_mtx, task_info, in_sch_info, ofp, sch_list,
               sch_file, dep_file, seed)
int adj_mtx[SIZE+1][SIZE+1]; /* adjacency matrix */
struct TASK task_info[SIZE+1]; /* task information */
struct IN_SCH_INFO *in_sch_info; /* task system information */
FILE **ofp; /* pointer to the output file */
int sch_list[MAX_PROCESSORS*SIZE+1]; /* used for static scheduling */
char sch_file[20]; /* file having static schedule */
char dep_file[20]; /* file having adjacency matrix */
double *seed; /* seed used by random number gen */
{
    FILE *tfp, /* file pointer to "sch_file" */
          *dfp; /* file pointer to "dep_file" */

    int length, /* length of a task */
        nprocs, /* number of processors */
        tasks, /* number of tasks */
        task_id, /* task id */
        succ_cnt, /* sucessor count */
        task_cnt, /* task count */
        i, j, /* loop counters */
        sch_list_cnt; /* static schedule count */

    char wait_type, /* user program or system command */
          in_out, /* inside/outside the scheduler */
          instr[220], /* variable to read line input file */
          *p, /* variable used in strtok() */
          sch[5]; /* temp variable */

    double rand; /* random number */

    /*
     * open static schedule file
     */
    if ((tfp = fopen(sch_file, "r")) == NULL) {
        printf("*** Error: can't open the schedule data file ***\n");
        exit(1);
    }

    /*
     * open adjacency matrix file
     */
    if ((dfp = fopen(dep_file, "r")) == NULL) {
        printf("*** Error: can't open the dependency data file ***\n");
        exit(1);
    }

    /*
     * store number of processors, number of tasks, in_out and wait type in
     * local variables.
     */
    nprocs = in_sch_info->nprocs;
    tasks = in_sch_info->tasks;
    in_out = in_sch_info->in_out;
    wait_type = in_sch_info->wait_type;

    /*
     * initialize variables
     */
    task_id = 0;
    sch_list_cnt = 0;

```

```

/*
 * read static schedule from the static schedule file and store it in
 * "sch_list" structure, the information in "sch_list" will later be used
 * for the static scheduling of the task system on requested processors.
 * Following code also store length of the task in task information structure,
 * i.e., "task_info".
 */
fgets(instr, 200, tfp);
while (!feof(tfp)) {
    p = strtok(instr, " ");
    length = atoi(p);
    sch_list_cnt++;
    if (length >= 0) {
        task_id++;
        sch_list[sch_list_cnt] = task_id;
        task_info[task_id].length = length; /* store length of task */
        /*
         * if wait_type is "X" then randomly assign "W" or "S", this shows
         * that the task system has mixture of 'S' and 'W' tasks.
         */
        if (wait_type == 'X') {
            rand = random(seed);
            if (rand > 0.50)
                task_info[task_id].type = 'W';
            else
                task_info[task_id].type = 'S';
        }
        else
            task_info[task_id].type = wait_type;
        task_info[task_id].in_out = in_out;
        /*
         * Depending on the wait_type of the task, copy the task which will
         * be executed.
         */
        switch (task_info[task_id].type) {
            case 'S' : strcpy(task_info[task_id].task, SYSTEM_COMMAND);
                        break;
            case 'W' : strcpy(task_info[task_id].task, USER_PROGRAM);
                        break;
        }
    }
    else
        sch_list[sch_list_cnt] = 0;
}

/*
 * read static schedule from static schedule file and store it in
 * "sch_list" structure, the information in "sch_list" will later be used
 * for the static scheduling of the task system on requested processors.
 * Following code also store length of task in the task information structure,
 * i.e., "task_info".
 */
do {
    p = strtok('\0', " ");
    if (p) {
        length = atoi(p);
        sch_list_cnt++;
        if (length >= 0) {
            task_id++;
            sch_list[sch_list_cnt] = task_id;
            task_info[task_id].length = length;
            /*
             * if wait_type is "X" then randomly assign "W" or "S", this shows
             * that the task system has mixture of 'S' and 'W' tasks.
             */
            if (wait_type == 'X') {
                rand = random(seed);
                if (rand > 0.50)
                    task_info[task_id].type = 'W';
                else
                    task_info[task_id].type = 'S';
            }
            else
                task_info[task_id].type = wait_type;
            task_info[task_id].in_out = in_out;
            /*
             * Depending on the wait_type of task, copy the task which will
             * be executed.
             */
            switch (task_info[task_id].type) {
                case 'S' : strcpy(task_info[task_id].task, SYSTEM_COMMAND);
                            break;

```

```

        case 'W' : strcpy(task_info[task_id].task, USER_PROGRAM);
                    break;
            }
        }
        else
            sch_list[sch_list_cnt] = 0;
    }
    while (p);
    fgets(instr, 200, tfp);
}
/*
 * read the adjacency matrix from the adjacency matrix file and put it into
 * "adj_mtx", also find number of successors for each task
 */
task_cnt = 0;
fgets(instr, 200, dfp);
while (!feof(dfp)) {
    task_cnt++;
    succ_cnt = 1;
    while (instr[succ_cnt] == ' ') succ_cnt++;
    while (instr[succ_cnt] != '\0') {
        sch[0] = instr[succ_cnt-1];
        sch[1] = '\0';
        adj_mtx[task_cnt][succ_cnt] = atoi(sch);
        succ_cnt++;
    }
    fgets(instr, 200, dfp);
}

/*
 * print the error message if the number of tasks in the adjacency matrix
 * is not equal to the number of tasks in the static schedule file
 */
if (task_cnt != tasks) {
    printf("***Error: Number of row %d of adjacency matrix\n",
           task_cnt);
    printf("      is less than the total number of tasks, i.e., %d.\n",
           tasks);
    exit(1);
}
sch_list[0] = sch_list_cnt;
}

/*
 *-----
 * task_schedul()
 *-----
 * This procedure statically schedules the tasks in the task system on the
 * requested number of processors.
 *-----
 */
void task_schedul(tasks, task_info, misc_info, sch_cnt, s_info, sch_list,
                  adj_mtx)
int tasks; /* number of tasks in task system */
struct TASK task_info[SIZE+1]; /* task information */
struct MISC *misc_info; /* progress information */
int *sch_cnt; /* number of tasks to be scheduled */
struct SCHE s_info[200]; /* schedule information */
int sch_list[MAX_PROCESSORS*SIZE+1]; /* static schedule of tasks */
int adj_mtx[SIZE+1][SIZE+1]; /* adjacency matrix */
{
    int task_id, /* task id */
        j, k, /* loop counter */
        nprocs, /* number of processors */
        processor, /* processor id */
        sch_list_cnt, /* number of entries in "sch_list" */
        sch_ndx; /* loop counter for fork() */

    sch_list_cnt = sch_list[0]; /* store number of entries in "sch_list" */
    nprocs = m_get_numprocs(); /* number of processors obtained */

    /*
     * get the processor id of the processor on which the procedure is being run
     */
    processor = m_get_myid();

    /*
     * if the processor id is nprocs-1 then run master clock on it
     */
    if (processor == (nprocs - 1))

```

```

    master_clock(misc_info);
else {
    /*
     * In current processor, execute the procedure task() and then execute
     * the procedure remove_dep()
     */
    for (sch_ndx=m_get_myid()+1; sch_ndx<=sch_list_cnt; sch_ndx+=(nprocs-1)) {
        /*
         * get task_id of the task to be executed in the current processor
         */
        task_id = sch_list[sch_ndx];
        if (task_id > 0) {
            /*
             * call procedure task() to execute the task
             */
            task(task_id, tasks, &task_info[task_id], misc_info, sch_cnt, s_info,
                adj_mtx);
            /*
             * remove all dependency of the current task from all the task in the
             * system, if they had some dependencies.
             */
            remove_dep(task_id, tasks, adj_mtx);
        }
    }
}

/*
 * -----
 * remove_dep()
 * -----
 * removes from the precedence graph, the current task as predecessor of any
 * task in the task system. This is done by replacing '1' with '0' in the
 * column of current task.
 * -----
 */
void remove_dep(task_id, total_tasks, adj_mtx)
int task_id, /* task id */
total_tasks, /* total number of tasks in task system */
adj_mtx[SIZE+1][SIZE+1]; /* adjacency matrix */
{
    int task_cnt; /* task count */

    /*
     * Replace '1' with '0' in the appropriate column of the current task
     */
    for (task_cnt = task_id+1; task_cnt <= total_tasks; task_cnt++) {
        if (adj_mtx[task_id][task_cnt] != DONE) {
            m_lock();
            adj_mtx[task_id][task_cnt] = DONE;
            m_unlock();
        }
    }
}

/*
 * -----
 * master_clock()
 * -----
 * This is the master clock.
 * -----
 */
void master_clock(misc_info)
struct MISC *misc_info; /* progress information */
{
    while ((*misc_info).clock_stop == FALSE) {
        m_lock();
        (*misc_info).clock++;
        m_unlock();
    }
}

/*
 * -----
 * task()
 * -----
 * Execute the task, first of all check to see all the predecessor have
 * finished the execution, then depending upon the value of wait_type and
 * in_out execute the task inside or outside the scheduler. Later on
 * gathers performance information and finly if all the tasks have
 * been executed stops the master clock.

```



```

-----
*/
void task(task_id, tasks, task_info, misc_info, sch_cnt, s_info, adj_mtx)
int task_id; /* task_id of current task */
int tasks; /* total number of tasks in task system */
struct TASK *task_info; /* task information */
struct MISC *misc_info; /* progress information */
int *sch_cnt; /* current schedule count */
struct SCHE s_info[200]; /* schedule information */
int adj_mtx[SIZE+1][SIZE+1]; /* adjacency matrix */
{
    long val; /* clock count for the current task */

    int i, /* loop counter */
        length, /* length of task */
        processor_id; /* current processor id */

    char task_com[60], /* task to be executed */
        temp[20]; /* temp variable */

    /*
     * Wait for predecessors to finish execution.
     */
    while (!pred_done(adj_mtx, task_id));

    /*
     * record the start time of execution.
     */
    m_lock();
    (*misc_info).progress[task_id].start_time = (*misc_info).clock;
    m_unlock();

    /*
     * get id of the current processor
     */
    processor_id = m_get_myid();
    (*task_info).processor_id = processor_id;

    length = (*task_info).length; /* get length of the current task */

    /*
     * Processes the task
     */
    switch ((*task_info).in_out) {
        /*
         * task is to be executed inside the scheduler
         */
        case 'I':
            switch ((*task_info).type) {
                /*
                 * Execute user program
                 */
                case 'W' : val = 0;
                    while (val <= (length * WHILE_CT))
                        val++;
                    break;

                /*
                 * Execute system command
                 */
                case 'S' : strcpy(task_com, "sleep ");
                    itoa(temp, length);
                    strcat(task_com, temp);
                    strcat(task_com, " ");
                    strcat(task_com, "du -s /t/tamir > du_out");
                    system(task_com);
                    break;
            }
            break;

        /*
         * task is to be executed outside the scheduler
         */
        case 'O': strcpy(task_com, (*task_info).task);
            strcat(task_com, " ");
            itoa(temp, length);
            strcat(task_com, temp);
            system(task_com);
            break;
    }
}
/*

```

```

    * Gather performance data
    */
m_lock();
(*sch_cnt)++;
s_info[*sch_cnt].processor = processor_id;
s_info[*sch_cnt].task      = task_id;
s_info[*sch_cnt].length    = length;
m_unlock();

/*
 * record end time of execution
 */
m_lock();
(*misc_info).progress[task_id].end_time = (*misc_info).clock;
m_unlock();

/*
 * Stop the master clock when all the tasks have been executed
 */
if (*sch_cnt == tasks) {
    m_lock();
    (*misc_info).clock_stop = TRUE;
    m_unlock();
}
}

/*
-----
 *                                     pred_done()
-----
 * Checks the adjacency matrix to see all the predecessors of current task
 * are done with execution.
-----
 */
int pred_done(adj_mtx, task_id)
int adj_mtx[SIZE+1][SIZE+1], /* adjacency matrix */
    task_id;                 /* task id */
{
    int task_cnt,             /* task count */
        pred_dn;             /* if TRUE, the predecessor is done with exec.*/

    /*
     * initialize the variables
     */
    pred_dn = TRUE;
    task_cnt = 1;
    /*
     * Check for predecessors to be done
     */
    while ((task_cnt < task_id) && (pred_dn)) {
        if (adj_mtx[task_cnt][task_id] != DONE)
            pred_dn = FALSE;
        task_cnt++;
    }

    if (pred_dn)
        return(TRUE);
    else
        return(FALSE);
}

/*
-----
 *                                     initialization()
-----
 * Initialize master clock, task information data structures and schedule
 * count.
-----
 */
void initialization(task_info, sch_cnt, misc_info)
struct TASK task_info[SIZE+1]; /* task information */
int *sch_cnt;                  /* schedule count */
struct MISC *misc_info;        /* progress information */
{
    int i;

    /*
     * initialize schedule count
     */
    *sch_cnt = 0;
}

```

```

/*
 * initialize master clock
 */
(*misc_info).clock = 0;
(*misc_info).clock_stop = FALSE;

/*
 * initialize task information
 */
for (i=1; i<=SIZE; i++)
    init_task_info(&task_info[i]);
}

/*
-----
 *
-----
 * init_task_info()
-----
 * Initialize task information
-----
 */
void init_task_info(task_info)
struct TASK *task_info; /* task information */
{
    /*
     * initialize processor id to 0
     */
    (*task_info).processor_id = 0;
}

/*
-----
 *
-----
 * itoa()
-----
 * convert an integer to a character string, i.e., 10 to '10'
-----
 */
void itoa(str_id, task_id)
char str_id[20]; /* character string of the integer */
int task_id; /* task id */
{
    int i,
        count,
        sw_val,
        temp_id;
    char temp_ch[20];

    count = 0;
    temp_id = task_id;
    while ((temp_id / 10) != 0) {
        sw_val = temp_id % 10;
        switch(sw_val) {
            case 0: temp_ch[count] = '0';
                     break;
            case 1: temp_ch[count] = '1';
                     break;
            case 2: temp_ch[count] = '2';
                     break;
            case 3: temp_ch[count] = '3';
                     break;
            case 4: temp_ch[count] = '4';
                     break;
            case 5: temp_ch[count] = '5';
                     break;
            case 6: temp_ch[count] = '6';
                     break;
            case 7: temp_ch[count] = '7';
                     break;
            case 8: temp_ch[count] = '8';
                     break;
            case 9: temp_ch[count] = '9';
                     break;
        }
        count++;
        temp_id = temp_id / 10;
    }
    sw_val = temp_id % 10;
    switch (sw_val) {
        case 0: temp_ch[count] = '0';
                 break;
        case 1: temp_ch[count] = '1';
                 break;
    }
}

```

```

case 2: temp_ch[count] = '2';
        break;
case 3: temp_ch[count] = '3';
        break;
case 4: temp_ch[count] = '4';
        break;
case 5: temp_ch[count] = '5';
        break;
case 6: temp_ch[count] = '6';
        break;
case 7: temp_ch[count] = '7';
        break;
case 8: temp_ch[count] = '8';
        break;
case 9: temp_ch[count] = '9';
        break;
}
count++;

for (i=0; i<count; i++)
    str_id[i] = temp_ch[count - i - 1];
str_id[count] = '\0';
}

```

```

/*
-----
*
*                               rand_gene()
*-----
* Random number generator. Generates a number between 1 and 0.
*-----
*/
double random(seed)
double *seed;
{
    double rand,
           lo,
           hi,
           test;
    int    tmp_int;

    tmp_int = *seed/q;
    hi      = tmp_int*(1.0);
    lo      = *seed - q*hi;
    test    = a*lo - r*hi;
    if (test > 0.0)
        *seed = test;
    else
        *seed = test + m;
    rand = *seed/m;
    return(rand);
}

```

VITA

Amir Ali Thobani

Candidate for the Degree of
Master of Science

Thesis: SCHEDULING OF SETS OF TASKS ON SEQUENT SYMMETRY
S/81: AN EMPIRICAL STUDY

Major Field: Computer Science

Biographical:

Personal Data: Born in Karachi, Pakistan, December 13, 1965, son of
Abbas Ali Thobani and Sultana A. Thobani.

Education: Graduated from D. J. Government Science College,
Karachi, Pakistan, in October 1984; received Bachelor of
Engineering Degree in Computer Systems Engineering from
N.E.D. University of Engineering and Technology at Karachi,
Pakistan in October 1990; completed the requirements for the
Master of Science degree in Computer Science at Oklahoma
State University in July 1993.

Professional Experience: Graduate Research Assistant, University
Computer Center, Oklahoma State University, May 1991 to July
1993.